Computer Science

COMPILING RECURRENT AND IRREGULAR SERIAL CODE
FOR HIGH PERFORMANCE COMPUTERS

ANWAR MOHAMMED GHULOUM
November 4, 1996
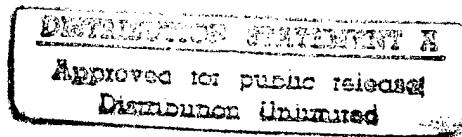CMU-CS-96-200

Carnegie
Mellon

COMPILING RECURRENT AND IRREGULAR SERIAL CODE
FOR HIGH PERFORMANCE COMPUTERS

ANWAR MOHAMMED GHULOUM
November 4, 1996
CMU-CS-96-200

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science in the School of Computer Science of Carnegie Mellon University, 1996.*   DTIC QUALITY INSPECTED 2

Thesis Committee:
Allan Fisher, Chair
Guy Blelloch
Thomas Gross
P. Geoff Lowney, DEC

19970702 062

Carnegie Mellon

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Compiling Recurrent and Irregular Serial Code for High Performance Computers

ANWAR M. GHULOUM

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

_____   November 4, 1996

THESIS COMMITTEE CHAIR   DATE

_____   1/16/97

DEPARTMENT HEAD   DATE

APPROVED:

_____   1/21/97

DEAN   DATE

For Leith

Acknowledgments

I owe a debt of gratitude to my advisor, Allan Fisher, for his contributions to this work and his support throughout the years. I would also like to thank my thesis committee members, Guy Blelloch, Thomas Gross, and Geoff Lowney for their guidance.

The Fx compiler group provided the software platform upon which this thesis built. Thomas Gross, Dave O'Hallaron, and Guy Blelloch provided me with the computing resources I needed at various times throughout this work. Jaspal Subhlok assisted me at times with compiler issues and questions.

A number of colleagues of mine deserve special recognition. Juan "Pablo" Leon, in reviewing this document (as well as countless other documents and presentations), has been so generous with his time and critical thought processes that I cannot thank him enough in this small space. Marco Zagha provided important insights into code generation issues for the C90. Anyone who ever attended a practice talk of mine or answered a question through any electronic media also has my gratitude.

My mother, Sandra, and my father, Mohammed, have always expressed confidence in me when I lacked it and support when I needed it. I thank them both enormously for this.

Above all, I would like to thank my wife, Elham, for her support and love throughout this long and difficult process.

Table of Contents

List of Figures

x

Chapter 1

Introduction

High performance computers are difficult to program. Tapping the potential of these systems requires that careful attention be paid to many system parameters and constraints, usually many more than lower performance systems. The skills acquired by the programmer to effectively program and tune the performance of her application on a particular system is often not applicable to another high performance system architecture. Higher level models are often tuned to particular class of architecture, with particular computation, memory access, and/or communication pattern characteristics and costs. Unless users are given adequate tools to program such systems, the potential gap between peak and realized performance will grow as systems increasingly rely on parallelism for higher performance.

Clean and coherent parallel programming models, whether manifested in library routines or new languages, have difficulty attaining widespread acceptance. Efforts to integrate high performance language primitives into serial languages have also met with limited success. Such efforts, in deference to the relative inertia to change in programming language usage, try to exploit the mainstream prevalence of imperative, serial languages. This often results in confusing and underpowered programming tools. Easy, portable, efficient, and widely used high performance programming is still an elusive goal.

Serial programming languages, like C and Fortran, satisfy three desirable properties for an effective programming tool. They embody a simple and intuitive model of computation. Compilers for such languages are available in a wide range of hardware and software operating environments. Finally, they dominate software development in all but a few niches. The only place they fall short

is in efficiently capitalizing on available parallelism. Great strides have been made in parallelizing loop nests and exposing instruction-level parallelism, however, there is still a big difference between achievable performance through careful hand tuning and performance through compilation.

A major shortcoming of existing compiler optimizations and automatic parallelization techniques is the limited reasoning mechanisms and models compilers rely heavily on. Higher level models and reasoning mechanisms have not been explored to enough depth thus far. For example, there are many examples of parallelizable loops which seem to be inherently serial when examined using only data dependence information, which parallelizing compilers rely heavily on. The use of perfect data dependence information is still a conservative method of determining whether a loop is parallelizable. There are alternative ways of characterizing or modeling serial code which simplify of the discovery of parallelism.

This dissertation uses existing models of control and data dependence and flow as a starting point for investigating one new analysis model. We focus on an important problem in automatically parallelizing serial code: recurrent and irregular code. Recurrent and irregular code play an important role in many scientific codes, as well as many sorting, geometric, and symbolic applications. No general techniques existed prior to this dissertation for automatically parallelizing such code. The extent of current techniques to deal with recurrent loops in automatic parallelization is to simply search for 'familiar' patterns of common loops in the source code. This dissertation will demonstrate, through a relatively simple semantic interpretation of loop bodies, the added power of a small amount of higher-level reasoning in an automatically parallelizating compiler.

1.1 Programming High Performance Computers

Computers rely increasingly on multiple processing elements, wider instructions, or deep pipelines to increase computational throughput and/or hide memory system latency. One example is Cray's multiple vector processor architecture, which is comprised of relatively few processors with deep pipelines, vector registers, chaining support, and multiple, shared memory banks. Distributed memory multiprocessor machines, such as the iWarp [22], Intel's Paragon, Cray's T3d, Thinking Machine's CM5 [84], among others, are typically comprised of processing elements and their own memories coupled by an interconnect network. Shared

memory multiprocessors rely on hardware (and sometimes software) mechanisms to give the programmer the illusion that the computer's memory is shared by all processing elements. Current and future generations of scalar processors rely increasingly on multiple instruction issue or long instruction words to keep multiple function units busy.

Mechanisms by which these processors use memory, communicate, and otherwise synchronize vary drastically. It is impractical to expect an application builder, especially one who is not a computer scientist specializing in systems, to explicitly control such mechanisms. For example, handling the details of message passing in a distributed memory system can be quite complicated. At a low level, the must explicitly manage message buffers and eliminate potential race conditions. At a higher level, the programmer has to explicitly manage the distribution of data and computation. These tasks are both error prone and cumbersome, in addition to requiring a good deal of expertise.

Programming tools such as software libraries, compilers and languages can simplify some of this complexity. High level parallel languages seek to provide intuitive mechanism for expressing parallel computation. In some cases, an existing serial language is extended with parallel primitives; for example, as in Fortran (HPF [46]), C (Split-C [29], C* [74][45]), C++ (Compositional C++ [26]), and Lisp/Scheme (Multilisp [40] , *Lisp [54]). In other cases, a new language is invented with the goal of easing the expression of parallelism; for example, NESL [18] and Linda [25], among others.

Other approaches provide control of parallelism through preoptimized libraries. Some libraries provide low-level support for expressing parallelism through message passing or shared memory synchronization primitives, along with support for threading. Other libraries provide data structures and routines for supporting parallel computation. Libraries such as CVL [17], CMMD [85], MPI [60], Nexus [37], Paris [83], and PVM [38] all seek to introduce support for parallelism with library calls linked into the application. The level at which parallelism is supported ranges from high-level operations on parallel object to low-level thread management, message passing and synchronization primitives.

Compilers are critical tools for translating high-level serial and parallel languages to machine specific code. Compiler translation and optimizations help to mitigate or hide the details of dealing with constraints on system resources and the low-level management of parallelism through synchronization. This is true of both compilers for explicitly parallel languages,

3

where some of these constraints may be implicitly defined in the language constructs, and automatically parallelizing compilers for serial languages. However, in the serial case, the compiler must contend with enforcing the more difficult semantic constraints imposed on it the source language.

This dissertation is concerned with compilers for programming parallel computers. In addition to exploring a different compiler model for computation, this research aspires to enable a programmer to write programs for high performance computers using the algorithms and structures they have become accustomed to working with, rather than forcing them to work around the tools by contriving new programming styles.

1.2 Automatic Parallelization[1]

A large body of research has been done and is ongoing in automated parallelization of serial source code. There are several rationales for this approach:

Ease of use: Serial languages have a naturally intuitive execution model. They are typically the languages people are most comfortable using. Widespread acceptance of serial languages, though certainly not proof of the superiority of such languages, makes them the most accessible mechanism by which to program parallel systems.

Portability: Compilers for serial languages like C are available for virtually every computer manufactured today. Building compiler technology to automatically parallelize a serial language does not lock the application builder into particular architecture or operating system.

Existing Code Base: There are many programs, so-called "dusty decks", that individuals and companies are reliant on and have invested much into developing and maintaining. The man-hours and expertise necessary to rewrite these programs using new parallel constructs or languages are potentially enormous.

Programmer Base: There is a good deal of inertia and apathy toward the adoption of new languages and programming paradigms in the programmer community.

---

1. At this point, we cease to discuss scalar compilation issues, such as instruction-level parallelism (ILP) and focus on data and task parallelism. However, this is not meant to imply that techniques here are not exploitable by compilers seeking to increase ILP.

1.2.1 Current Limitations

Compilers which automatically parallelize serial code traditionally rely on dependence and alias analysis on loop nests to expose potentially parallelizable regions [95]. Dependence analysis and array data-flow analysis provides detailed flow information for loops with indexed array expressions. Loop nests are natural candidates for parallelization because of replication of computation in the loop body over potentially many iterations. The goal of the compiler is to relax serial execution semantics for parallelization as long as various dependence constraints are not violated. Typically this is manifested in the compiler only parallelizing those regions for which there are cycles of flow dependence links with loop-carried dependence(s).

The compiler can do much in the way of loop nest manipulation to expose loops as candidates for this kind of parallelization. Recently, a resurgence of work in loop transformation frameworks [91] has created a well defined theory of loop nest transformation to expose parallelism, improve locality, and perform other optimizations. However, these techniques are limited to loop nests whose loops bounds and array index expressions are *analytically manageable* by the compiler. This typically means that the loop bounds must either be constant, or linear in surrounding loop indices and variables in scope.

A shortcoming of relying exclusively on dependence information (even if that information is perfect) is that the nature of the parallelism it will expose is too conservative. There are many useful computations for which parallel regions cannot be exposed through dependence based parallelization, but are still parallelizable through algebraic transformation. The prevailing mechanism that compilers use to deal with such regions is through *pattern matching*. For example, consider the following loops:

```
do i = 1, n
   a = a + b(i)
enddo

a(1) = b(1)
do i = 2, n
   a(i) = a(i-1) + b(i)
enddo
```

The first loop sums all elements of the array b into a scalar variable a and is called a *reduction*. The second loop performs essentially the same computation but stores all partial (or prefix) sums and is called a *scan* or *prefix-sum*.

In general, reductions and scans can be parallelized if the operator used is associative (e.g. addition). However, dependence information for these loops would include parallelization inhibiting dependences. So, rather than develop more general alternatives to dependence-based approached, the compiler writer, recognizing that this is still a parallelizable computation, will often get around this problem by hard coding a search for that pattern of loop in their compiler.

This approach severely limits the ability of the compiler to generally parallelize the many variations of these *recurrences*. There are other useful parallel building blocks expressible through such recurrent loops, such as combing-send, multiprefix, and list ranking based algorithms [71][77]. The outlook for automatically parallelizing these is even bleaker in current compilers.

Support for the expression of nested parallelism is critical to simplifying the parallelization of many algorithms. For example, a matrix can be viewed as an array of rows, which are also arrays, or an array of columns. Thus, any operation on the entire matrix (such as matrix-vector multiplication) would be most concisely expressed using nested control structure. For example, for a dense matrix-vector multiplication, the following code is typical:

```
do i = 1, rows
    y(i) = 0
    do j = 1, columns
        y(i) = y(i) + matrix(i,j)*vec(j)
    enddo
enddo
```

However, this prototypical matrix-vector multiply code performs poorly for sparse matrices because of the limited amount of *useful* work it will accomplish relative to the work it is performing in treating the matrix as if it were dense. One possible representation for sparse matrix-vector multiply [32] gives the following loop nest:

```
do i = 1, rows
    y(i) = 0
    do j = pntr(i), pntr(i+1)-1
        y(i) = y(i) + spmatrix(j)*vec(cols(j))
```

```
      enddo
enddo
```

One problem here is that this code is not amenable to any loop nest transformations to expose outer loop SIMD style parallelism. However, the dependence based approach can expose more general parallelism in the outer loop, though this can have other shortcomings, which we will discuss further in chapter 5. Any finer grained parallelism exposed by parallelizing or vectorizing the inner loop may be limited by the sparsity of the matrix.

Another kind of programming style which proves problematic for existing compiler technology is divide-and-conquer. In this case the nesting control structure is not in loops, but rather in recursive subroutine calls. Support for automatically parallelizing this kind of irregularly nested parallelism in serial code efficiently is virtually non-existent.

1.2.2 Current Approaches

The widespread availability of loop kernel suites for evaluation of compilers has constrained those recurrences that many commercial compilers parallelize to those present in such suites. Superficially, this seems to be a reasonable approach since those kernels are representative of most of the recurrent loops one find in scientific code. Evaluating of a new compilation technique is difficult against such a stacked deck.

The flaw of the pattern matching approach to parallelizing recurrent loops is the fixation on particular forms of recurrences, rather than any meaningful semantic models. The algorithm that a programmer may hypothetically employ for a matrix multiplication may (arguably) be predictable, but the particular form or syntax may not be predictable. In other words, the reality is that most people who program may write similar code, but with slight variations. Pattern matching locks out many reasonable and innocuous variations. These loop kernel suites can be used to illustrate this problem.

We have tested our compiler on the Argonne loop suite's [55] recurrent loops. The range of recurrence types in this suite is inclusive of many in other suites. To test the relative robustness of the underlying detection technique, we have also compiled a set of the Argonne loops with a slight perturbation. The variation we chose was to add another variable to carry values across loop iteration, creating a breakable coupled recurrence. For the two recurrences of the previous section, the transformed code would appear as:

```
carry = a
do i = 1, n
   a = carry + b(i)
   carry = a
enddo

carry = a(1) = b(1)
do i = 2, n
   a(i) = carry + b(i)
   carry = a(i)
enddo
```

We have constructed a simple graph comparing our compiler on the original and transformed code, the Cray Research Fortran Compiler (CF77) and the Applied Parallel Research Forge/ DM Fortran compiler [8] for the Argonne loops and the varied loops in figure 1.1. Our compiler compares favorably to both compilers with the basic set of Argonne loops. The one case in which we do not parallelize where CF77 does (Loop 332) involves a loop exit statement. This is not a limitation of the underlying model and analysis of our technique. Rather, it is a limitation of the front end to the recurrence parallelization technique and our choice or primitive templates for code generation.

The disparity in performance with the varied loop set is even more significant. The Cray CF77 compiler can no longer parallelize over half of the loops that it could with the variation, while the Forge compiler is unable to parallelize any of the loops. This confirms the sensitivity of the other compilers on the syntactic quality of the source code. Our technique's performance is undisturbed by the variation.

1.2.3 Frequency of Recurrence and Irregularity

The example in section 1.4 demonstrates the power of recurrences, especially when one can compile irregular control structure in conjunction with the recurrences. However, an important consideration for the automatic parallelizing compiler community is the presence of recurrences in existing scientific codes.

The graphs in figure 1.2 illustrate the percentage of time spent in recurrent or irregular code in the whole program benchmark suites PERFECT [88] and NAS [75][2], respectively. Even the

---

2. Our compiler can profile and instrument programs to measure the dynamic impact of recurrent and irregular codes. Details on how this profiling is done is presented in chapter 7.

Loops Parallelized from the Argonne Loop Suite (18 recurrent loops)
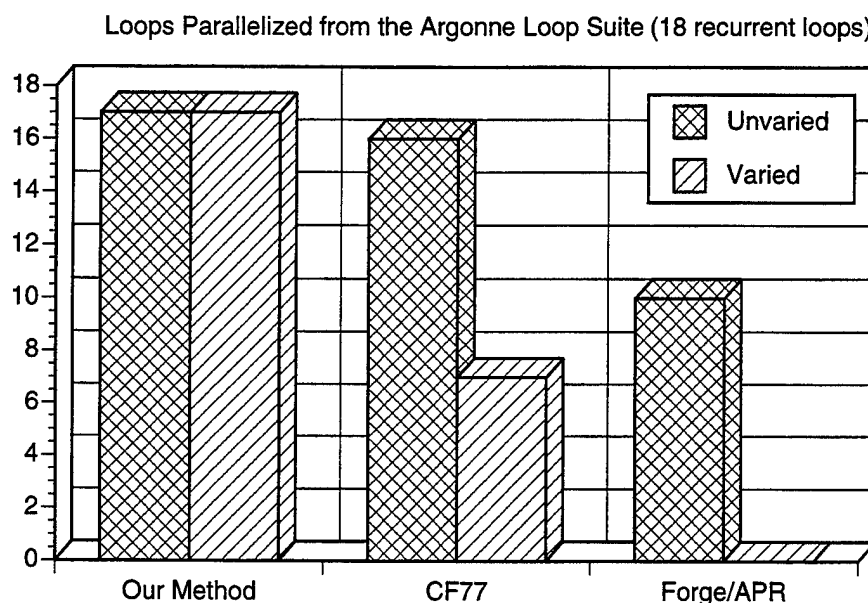
Figure 1.1 Recurrent loops parallelized from the Argonne Loop Suite.

Proportion of Time Spent in Recurrent Loops
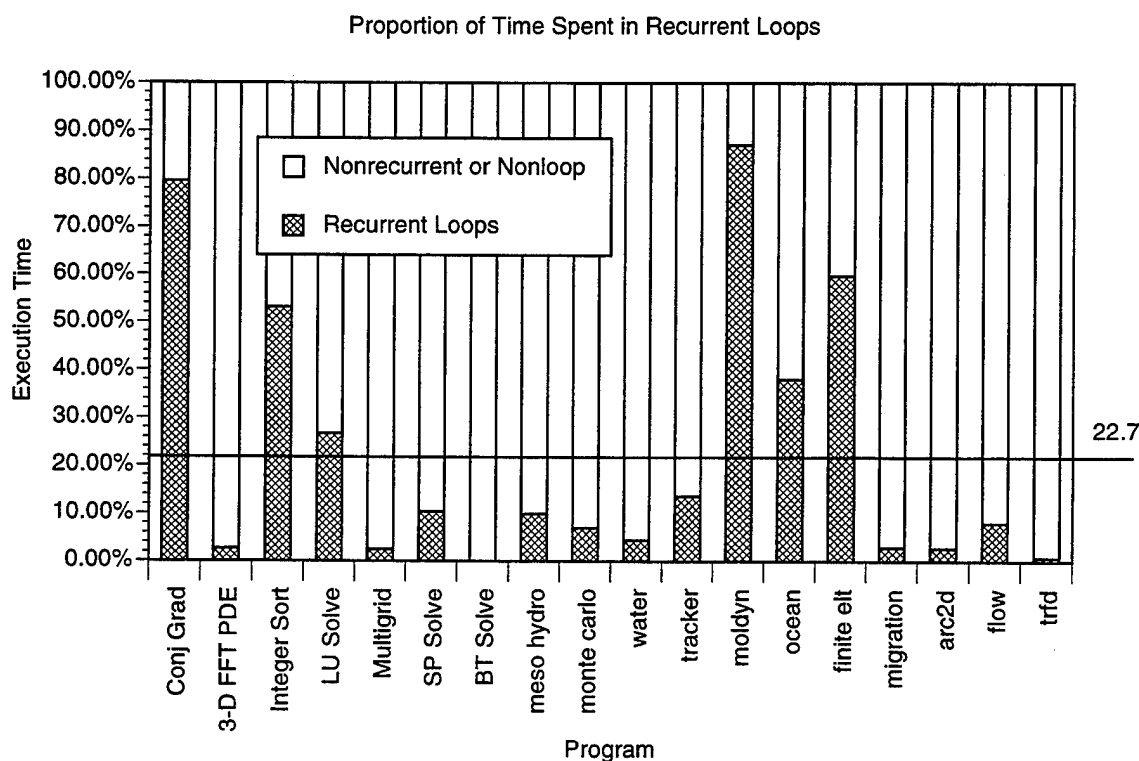
Figure 1.2 Dynamic profile of percentage of time spent in recurrent loops.

less significant numbers in this graph have a big impact on maximum speedup if recurrent loops are not parallelized, illustrated by the simple application of Amdahl's Law in figure 1.3.

9

Speedup Limitations to Parallelization by Restricting Recurrences
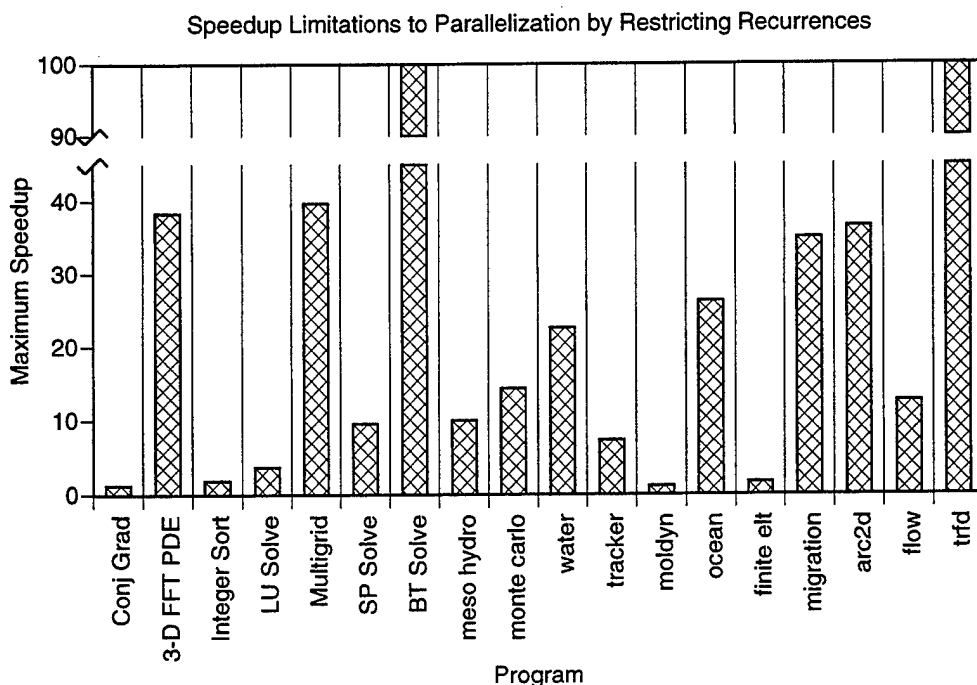
Figure 1.3 Limitations on achievable speedup due to serialization of recurrent loops.

Note that this only measures the impact of recurrent loops, and does not include information about other parallelization inhibiting code structures and overhead. This measure of potential speedup is extremely optimistic in that regard. Furthermore, many loop kernel oriented benchmark suites are comprised of a significant number of recurrent loops [34][55]. These figures justify efforts to parallelize such loops, but do not give any indication about which techniques should be employed. However, taken together with the poor performance of commercial compilers performing pattern matching on the loops in section 1.2.1, along with the more complex control contexts in which simple recurrences can occur, this provides strong motivation to improve on current techniques for parallelizing such code.

1.2.4 Our Approach

The success of research in developing expressive and efficient associativity-based parallel primitives, such as reduction, scan, combining-send, and their segmented variants, is evident in its broad application to the parallelization of scientific code, sorting, and other recurrent and irregular applications [15][20][77]. The essence of this success is that these primitives gracefully encapsulate computation patterns that are both complex and efficiently mappable to hardware.

Prior work in this area focused on exploring the expressiveness of the primitives. Support for these primitives is considered by many critical to the success of various strategies for supporting parallelism. Work has also focused on mapping them efficiently to hardware, typically augmenting an existing programming language or a parallel language with their expressiveness. This dissertation leverages off this work and automatically detect and extract those patterns of serial code that can be recast into these powerful parallel primitives.

The cornerstone of this dissertation is a general and robust technique for analyzing recurrent loops and automatically deriving reductions, scans, and other recurrent primitives over arbitrary operators. The technique does not rely on pattern matching to find 'known' recurrences; rather, it relies only on observations of desirable properties, such as associativity and efficiency, with the concomitant greater breadth of applicability. The technique has been integrated into a parallelizing compiler for serial Fortran.

This dissertation also addresses the problem of compiling nested parallelism as expressed in irregular loop nests and divide-and-conquer style recursion. Two transformations, loop flattening and a variant of loop embedding for recursive subroutines, effectively enable the recognition of segmented variants of the basic recurrent primitives. The essential idea is to transform the nested iterative or recursive structure into nested conditionals which the underlying recurrent loop analysis can successfully parallelize. These control structure transformations and the recurrence parallelization technique presented in this dissertation exhibit good synergy when applied to the aforementioned irregular algorithms.

The dissertation will discuss the rationale and design of these analyses and transformations. We will describe their implementation in a parallelizing fortran compiler and present the results of compiling both commonly used algorithms and code from scientific benchmark suites.

1.3 The Thesis

The thesis of this dissertation is that generally parallelizing recurrent and irregular serial code is possible and worthwhile in an automatic parallelizing compiler. In demonstrating this, the dissertation will include:

- A description of transformations to support parallelization of recurrent and irregular code.

- An implementation of those transformation in a parallelizing compiler.

- A demonstration of the power of these transformation on important algorithms and bench-mark programs.

1.4 Example: Quicksort

Code that is both recurrent and irregular occurs in many algorithms we are interested in compiling. The code for a quicksort is indicative of one such instance of a program. We list that code below, with annotations at points of interest (for brevity, we restrict ourselves to a non-stable version for sequences of non-repeating keys):

```
subroutine qsort(a,b,begin,end,n)
integer begin,end,n
integer a(n), b(n)
integer lower,upper, i, pivot

pivot = a(begin)
lower = begin
upper = end
do i = begin,end
    if (a(i) < pivot) then
        b(lower) = a(i)
        lower = lower + 1
    else
        b(upper) = a(i)
        upper = upper - 1
    endif
enddo
call qsort(b,a,begin,lower-1,n)
call qsort(b,a,upper+1,end,n)
end
```

1. Recurrence

2. Irregularity

3. Nested Control Structure

The first item of note in this subroutine is the presence of the recurrence in computing the monotonic induction variables [94] lower and upper. These can be computed independently of the data motion involved in partitioning the sequence. Having done this, the loop can be executed in two phases: compute lower and upper using a scan, then merge the result and use it as an index array to a permute. Alternative approaches might use packing operations to move the data.

12

The second item of note is that the loop bounds are symbolic values, about which the compiler has no knowledge. This means that the compiler has no way of knowing whether sufficient parallelism exists in the loop to justify a particular parallelization approach. Furthermore, it has no way of statically characterizing the workload distribution for the loop.

The third thing to notice is that the sequence is partitioned and recursed upon. This nested control structure essentially iterates over each partition, executing the body of the loop. Since the embedded computation is a loop with bounds which vary irregularly, we are confronted with the problem of essentially compiling nested irregular and recurrent computations.

Such combinations of recurrence and irregularity occur frequently in divide and conquer algorithms. Code to partition array based structures tend to be recurrent, while the partitions themselves tend to be constructed in an irregular fashion. In light of this, building recurrence parallelization techniques without regard to irregularity, or vice-versa, makes little sense.

1.5 New Compilation Techniques

1.5.1 General Recurrence Parallelization

A more general analytical approach is preferable to one in which the programmer must, as is often the case with ad-hoc approaches like source level pattern matching, expected to understand how the compiler works. This dissertation presents a new approach to parallelizing recurrences based on observations of desirable mathematical properties in the code. In this case, the desirable mathematical property is *associativity*. Discovering whether or not associative operators are present in loops with parallelization-inhibiting dependences opens up opportunities for using the parallel primitives we will introduce in the next chapter, which we will refer to as *recurrent primitives*.

Our compilation technique first finds associative operators in the recurrent loops. The compiler maps the recurrent loops to a particular recurrent primitive (i.e. reduction or combining send, etc.) by examining the traversal pattern of the computation on the target structure. For example, if the target structure is a scalar value, then the operation is a simple reduction.

13

This compilation technique forms the kernel of this research. To compile recurrent code nested in irregular loop nest or recursive subroutine calls, we layer control structure transformations which make the code more amenable to the application of this core analysis.

1.5.2 Irregular Control Structure

Many algorithms rely on the ability to partition the problem into smaller sizes or to traverse nested data structures. For example, code for a matrix operation would iterate over each row, in which it iterates over each column, or vice-versa. A natural way to express this is through nesting of loops which iterate over pointers in each dimension of the structure. Another useful example is that of a divide and conquer style algorithm which solves a problem by subdividing it into smaller tasks. The most prevalent and natural mechanism used to express these computation is through the use of divide and conquer style recursion, where the partitions of the problem are recursed upon, and then subdivided themselves, and so forth [4].

As previously mentioned, current compiler technology excels at parallelizing regular loop nests, i.e. those whose loop bounds, at their most complex, are linear in outer loop indices. However, for many problems, especially sparse matrix algorithms, the loop bounds are more complex. The loops bounds will vary arbitrarily, usually due to dynamically determined data values. The code in section 1.2.1 is an example of this kind of irregular control structure. The loops bounds inhibit parallelization via traditional techniques which expose all the parallelism available in regular loop nests.

Divide and conquer style recursion is currently not compiled effectively in serial code. Most compilers will try to parallelize the subroutine body. The problem is that at the algorithm progresses, the problem size decreases, decreasing the amount of parallelism available in the function body. While there is available inter-procedural parallelism in the recursive calls to the partitions of a newly subdivided problem, current compilers cannot exploit this.

In both of these cases, it is the control parallelism that is difficult to exploit. This dissertation presents mechanisms to exploit this control parallelism in a data-parallel context. The techniques presented have the added benefit of being perfectly compatible with the core recurrence parallelization technique in our compiler and the back-end compiler, which employs more traditional parallelization techniques.

14

1.6 Overview of the Dissertation

The primary goal of this work is to move beyond dependence analysis and pattern matching based approaches to automatic parallelization, focusing on parallelizing recurrent and irregular computation. The main claim of this dissertation is that recurrent and irregular code can be automatically mapped into useful higher level parallel primitives in a general and reliable manner. The primary contributions of this dissertation are:

- General and robust compilation techniques for automatically parallelizing recurrent loops into reduction, scan, combining-send, and multiprefix operations. These techniques are flexible enough to deal with other important components of the compilation process.

- Compilation techniques for parallelizing irregular loop nests. These techniques are fully compatible with both the underlying recurrence parallelization technique and traditional dependence based approaches.

- Compilation techniques for parallelizing divide-and-conquer style algorithms. These techniques are fully compatible with underlying techniques for parallelizing irregular loop nests, recurrent loops, and non-dependence loops.

In practical terms, these three contributions mean that difficult algorithm classes including sparse matrix operations, sort, and computational geometry, among others, may be automatically parallelized from their serial encodings. The evaluation of the effectiveness of these tools has three basic components:

- Profiling prevalent 'whole program' benchmark suites to get dynamic counts of the amount of time spent in recurrent loops. This will demonstrate that recurrences are a significant factor at run time to make a serious effort at parallelizing them.

- Comparing the robustness of our technique with commercial (typically, pattern matching) techniques with respect to collections of loop kernels and variations on them. This will demonstrate that pattern matching is a poor technique to use.

- Evaluating the performance of automatically parallelized code generated by our compiler on algorithmic techniques which were heretofore difficult to compile. This will demonstrate that our framework enables much more than merely the parallelization of a few kernels.

I will not explore the following in this dissertation at any great depth, since they fall outside the scope of this dissertation, though they do play important supporting roles in my work:

- Developing high performance implementations on the various parallel primitives we work with on various architectures. The primary focus of this dissertation is compiler analysis. We have relied on the work of others for my implementations of the various recurrent primitives in the code generation phase of the compiler.

- Developing new algorithms or a benchmark suite by which to evaluate compilers. We have relied on accepted benchmark suites and straightforward Fortran encodings of basic algorithms from texts. The collection of programs we compile may turn out to be useful benchmarking tools, but we make no claims in this regard.

- Developing decision mechanisms for deciding when to employ the transformations and analyses presented in this dissertation. We do not apply these technique blindly, however there is a body of work available and ongoing in profile-driven compilation and predicated execution of parallelized code [89].

1.7 Organization of the Dissertation

The rest of this document is structured as follows. Chapter 2 will discuss in more detail the recurrent primitives and code that we are interested in compiling. Chapter 3 will discuss the basic analysis for compiling recurrent loops. Chapter 4 will detail how to we integrate this into a compiler and general code. Chapter 5 will discuss techniques for compiling irregular loop nests. Chapter 6 will discuss techniques for compiling divide-and-conquer style recursion. Chapter 7 will discuss the compiler architecture. Chapter 8 will discuss the compilation results for the evaluation suite we have chosen for the compiler. Chapter 9 will discuss related work. Chapter 10 will discuss future work, contributions, and conclusions.

Chapter 2

Recurrent and Irregular Code and Primitives

This chapter reviews the various primitives and the types of serial code we are interested in compiling. In subsequent chapters, we will refer to these operations and their parallel implementations frequently, so it is worthwhile explaining them here.

Since the compiler is a fortran compiler, and our target is a shared memory vector multiprocessor, this chapter may specifically address issues pertaining to that context. However, the higher level issues of applicability of these primitives to various algorithms and their encoding is general. For example, code examples are presented in a pseudo-fortran, but are applicable to other serial, imperative languages like C.

'Recurrent parallel primitives' can be defined as parallel primitives whose natural serial encoding is through recurrent loops. Reduction, scan, combining-send, and multiprefix operations are all examples of such primitives. The primary goal of this section is to provide an overview of these primitives and their serial encodings. We will discuss these in some detail, but refer the reader to references for deeper discussions of their implementation on various architectures.

The recurrent primitives just introduced can be expressed serially in a variety of ways. The compilation techniques in this thesis target specific methods of expressing those primitives. Specifically, we are concerned with code written in imperative languages like C or Fortran, the latter of which the compiler is written for. Even with this constraint, there are a number of ways to encode many of these primitives. This section provides a breakdown of the various types of code we are interested in compiling and to which recurrent primitives they can compile.
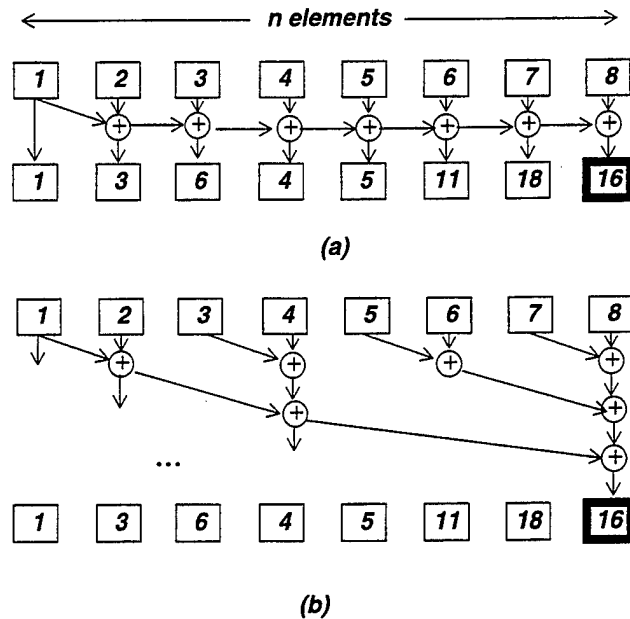
Figure 2.1 A (a) serial and (b) parallel reduction (bold boxes) and scan (all boxes).

Many parallel algorithms exist to perform these various recurrent primitives. They range from fairly high level to very architecture specific. This compiler targets the Cray C90 architecture, so this section will focus on algorithms specific to that genre of architecture. This work is largely derived from the work of others, and we note where we have made substantial changes. We refer the reader to the original work for a deeper treatment of these algorithms.

2.1 Reductions and Scans

2.1.1 Description

A reduction of a binary operator on an array[1] (or vector or collection) applies that operator across each element of the array and the running sum result of the operation. Figure 2.1a illustrates the flow of values in a serial reduction operation. If the operator reduced upon is associative, we can reassociate the operator applications so that some can occur in parallel. Figure 2.1b illustrates a reassociation in which the reduction takes $O(\log n)$ parallel steps, where n is the size of the array.

---

1. Since we compile Fortran, we will primarily discuss these primitives in the context of arrays.

A scan of a binary operator on an array applies that operator across each element of the array and the running sum, while retaining all partial sums. Scans are essentially identical to reductions in the serial case, and are very similar to reductions in the parallel case. The primary difference in the parallel case is that during the initial phase, which is essentially a reduction, partial results at each node are retained, and then propagated back downward. The performance of a parallel scan is thus within a constant factor of the performance of the parallel reduction.

The power of reductions and scans and their segmented variants was first fully explored by Guy Blelloch [15]. Work efficient parallel prefix-sum operations were first described by Stone [80]. Here is a simple example of their use in parallelizing the following basic linear recurrence:

```
do i = 2,n
   a(i) = b(i)*a(i-1) + c(i)
end do
```

This recurrence can be computed in a parallel scan of the following (associative) operator across the two arrays b and c:

$$(x, x') \oplus (y, y') = (xy, xy' + x')$$

The results of the scan can then be used to evaluate the result at each point in the scan. The result computation looks like:

```
tmp_pairs(2:n) = par_scan(f, a, b)
a(2:n) = first(tmp_pairs(2:n))*a(1) + second(tmp_pairs(2:n))
```

Where `first` and `second` select the first and second elements of a pair. In Fortran, this can be implemented by adding a dimension of size 2 and then using indexing to select the value.

Performing *n*th order reductions and scans, linear recurrences, and so on, are trivial with parallel scans and reductions. All the programmer and compiler need is to find an appropriate associative operator, if one exists.

2.1.2 Serial Encoding

As previously discussed a serial computation, typically a loop, that is intrinsically a recurrent operation is what we classify as a recurrence. That is, such a segment of code is a candidate for parallelization by using reduction, scan, combining-send, and multiprefix operations. For the moment, consider only non-nested recurrent loops as our targets, since we will the next type of serial code we discuss will subsume all nested recurrent loops. The code examples presented thus far are good indications of the type of code we generally target. The general form is:

```
do <some_range>
   recur_value(<current>) = f(recur_value(<previous>))
end do
```

A more operational definition for a compiler would be those loops which have loop carried cycles of flow and control dependences. The latter condition accounts for the presence of conditional constructs, for example, in the following loop:

```
do i = 1, n
   if (a .lt. b(i)) then
      a = b(i)
   end if
end do
```

2.1.3 Implementation

The scheme we use for performing reductions and scans is based on a simple block-wise decomposition of the array, as in figure 2.2. The first phase computes P partial sums on the P processors. All computation in this phase is local. The following fortran pseudocode, with the inner loop parallelized, illustrates this operation (this implementation is mostly based on [19]):

```
do i = 2, rows
   do j = 1, cols
      prefix((i-1)*cols + j) = prefix((i-2)*cols + j)
$                                      ⊕ val((i-1)*cols + j)
   enddo
enddo
```

The second phase serial sums the P partial sums and propagates them across the processors. A reduction can stop here.
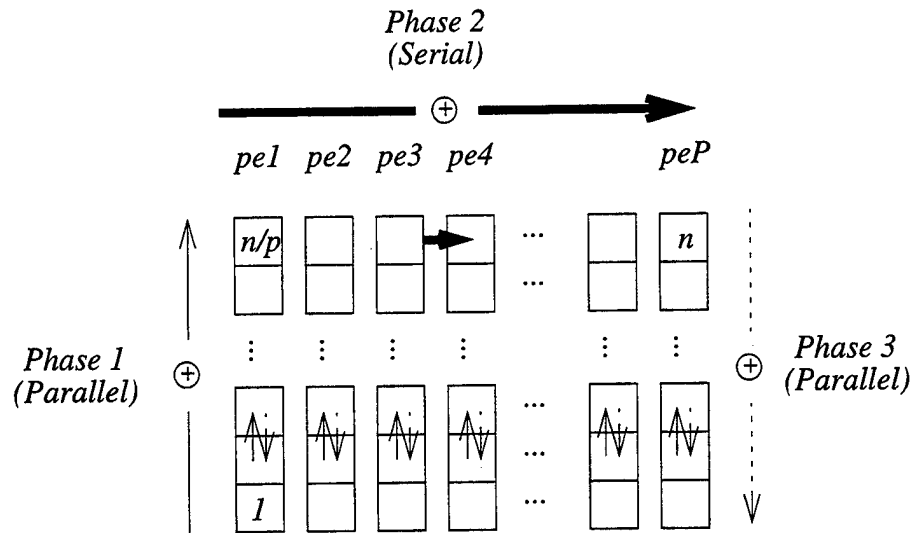
Figure 2.2 A vectorizable scheme for executing a reduction and scan.

```
do j = 2, cols
    colsums(j) = colsum(j-1)  ⊕  prefix((rows-1)*cols+j-1)
enddo
```

Phase 3 is reserved for scans and segmented reductions and scans. In a scan, phase 3 propagates the partial sums computed in phase 2 to all the partial sums computed in phase 1.

```
do i = 1, rows
    do j = 2, cols
        prefix((i-1)*cols + j) = colsum(j)
$                                      ⊕ prefix((i-1)*cols + j)
    enddo
enddo
```

In a segmented reduction or scan, the values are propagated only to those partial sums from phase 1 involved in segments which cross processor boundaries.

We use approximate versions of this basic computational template on both the iWarp and Cray C90. For the C90, one can view each slice of the vector register bank as a virtual processor on which this algorithm runs. On a single vector processor, the value P is the size of the vector register length. On multiple vector processors, the value P is the size of the vector register length multiplied by the number of processor.
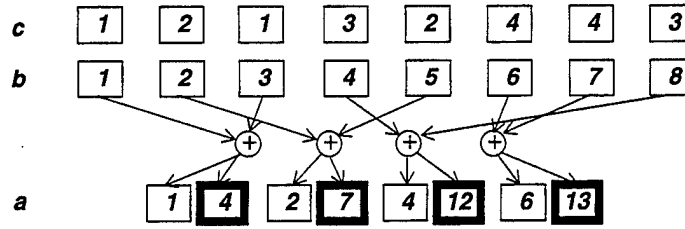
Figure 2.3 Combining-send (bold boxes) and multiprefix operation (all boxes).

2.2 Combining-send and Multiprefix

2.2.1 Description

Combining-send operations are essentially permutations with write conflicts resolved by a combining operator. Figure 2.3 illustrates such an operation. The computation can be parallelized if the combining operator is associative. It can be parallelize more effectively if the combining operator is commutative, that is if it does not matter in what order the operations are applied. However, we are interested in the more general case, and consider commutativity a simple optimization of that. In the general case, the starting point is to find an associative operator.

The following serial weighted histogram is an example of where a parallel combining send can be used:

```
do i = 1, n
   a(c(i)) = a(c(i)) + b(i)
end do
```

Obviously, the combining operator here (addition) is commutative as well as associative. Combining sends can be used to express more powerful operations, such as the following version of sparse matrix-vector multiplication (note the difference in representation from the last section):

```
do i = 1, N, 1
   do k = pntr(i), pntr(i+1)-1, 1
      y(indx(k)) = y(indx(k)) + val(k)*vec(i)
   end do
end do
```

A multiprefix operation is very similar to a combining-send operation, except that partial results for each element of the target array are retained. Figure 2b also illustrates this operation. The relationship of a multiprefix operation to a combining-send is analogous that of a scan to a reduction. An example of a place where a multiprefix might be employed is the following code segment:

```
do i = 1, n
   a(c(i)) = a(c(i)) + b(i)
   d(i) = a(c(i))
end do
```

Here, the partial results are stored in the array d. The algorithm for computing a multiprefix operation is identical to that for implementing the generalized combining-send.

2.2.2 Serial Encoding

Section 2.1.2 described the general form of serial code for expressing recurrences, which include combining-send and multiprefix operations. The only difference is in the structure traversal pattern, which, in this case, is dependent on an array of indices.

2.2.3 Implementation

Our computational template for computing combining-sends (also called a *multireduce* operation) and multiprefix operations is based entirely on the work of Sheffler [76] (the code, illustrations, and terminology are all based on his implementations). The basic algorithm, illustrated in figure 2.4, is comprised of phases similar to those in the reduction and scan templates. Combining-send and multiprefix operations are essentially generalizations of reductions and scans: each element has a key or label and each element contribute its value only to computations involving elements with the same key. That is, each element sums its value to the value of each previous element which contains the same label.

The key idea is to preserve this property by building and maintaining a structure, called a *SPINETREE*, which links together elements with the same label. The array of values the operation is taking place on is decomposed into a rectangular shape. Each SPINETREE link in each row points to one element with the same label in the next row. The spinetree can be built using the following pseudo-fortran loops:

23
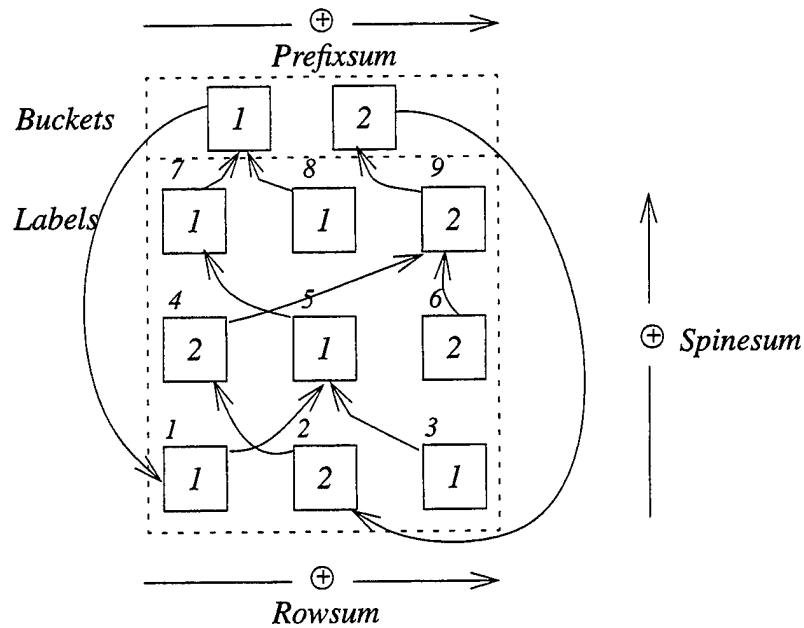
Figure 2.4 The SPINETREE structure and phases of generalized combining-send and multiprefix operations.

```
do j = rows, 1
    do i = 1, cols
        spine((i-1)*cols + j) = bucket(label((i-1)*cols + j))
        bucket(label((i-1)*cols + j)) = spine((i-1)*cols + j)
    enddo
enddo
```

The SPINETREE structure can be reused for each use of the same `label` array. The algorithm now progresses as follows. The first phase, the *rowsum* phase, sums each element's value in each column into the whatever element its SPINETREE link points to.

```
do j = 1, cols
    do i = 1, rows
        rowsum(spine((j-1)*cols + i)) =
$                          rowsum(spine((j-1)*cols + i))  ⊕
$                          value((j-1)*cols + i)
    enddo
enddo
```

The net effect is to sum each row, but only summing similarly labelled elements. The second phase, the *spinesum* phase, sums elements up the spine of each SPINETREE to create prefix sums for each label at the start of each row.
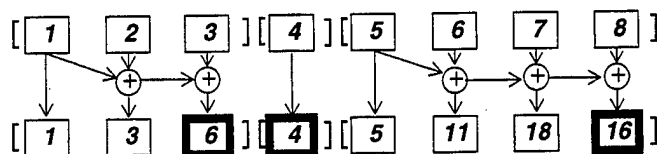
Figure 2.5 An inclusive segmented reduction (bold boxes) and scan (all boxes).

```
do j = rows, 1
   do i = 1, cols
      if (this_element_is_on_a_spine) then
         spinesum(spine((i-1)*cols + j)) =
$                            rowsum((i-1)*cols + j)  ⊕
$                            spinesum((i-1)*cols + j)
      endif
   enddo
enddo
```

This phase essentially ends the computation in the case of a combining-send, since the bucket structure will hold the total sums for each label (by adding the rowsum and spinesum). The final phase, the *prefixsum* phase, essentially incorporates the spinesum values into each element.

```
do j = 1, cols
   do i = 1, rows
      multi((j-1)*cols + i) = spinesum(spine((j-1)*cols + i))
      spinesum(spine((j-1)*cols + i))  ⊕ = value((j-1)*cols + i)
   enddo
enddo
```

2.3 Segmentation

2.3.1 Description

The real expressiveness of reductions and scans is evident when one adds the ability to 'segment', or arbitrary restart the computation and points in the sequence [14]. This essentially allows one to partitions the computation arbitrary. The typical scheme is to have some denotation of partition boundaries on the array, either in the form of a bit-vector, segment length array, or condition array. The reduction or scan occurs independently in each partition or segment, as illustrated in figure 2.5.

A nice feature of segmented reductions or scans is the flexibility they give the user or compiler in their implementation. One can either perform the operation within each segment in parallel perform each segment's parallel reduction or scan, or it can perform both simultaneously. Furthermore, segmented reductions and scans can be *flattened* so that the computation itself looks like non-segmented reduction or scan. Trade-offs in selecting one of these depend on multiple factors, such as the average size and variance of each partition and the communication/memory access overhead of the target system.

Since our compiler targets primarily the Cray C90, we choose to flatten these segmented operations wherever possible, to exploit the better load balancing characteristics and higher availability of parallelism.

The power of a segmented operation is evident in its use in parallelizing the following sparse matrix vector multiplication kernel:

```
do i = 1, N, 1
   y(i) = 0.0
   do k = pntr(i), pntr(i+1)-1, 1
      y(i) = y(i) + val(k)*vec(indx(k))
   end do
end do
```

These can be easily and effectively parallelized using a segmented reduction. (Chapter 5 will discuss this in more detail.)

A nice feature of segmented operations is the arbitrary partitioning. This enables their use in parallelizing a variety of problems which employ partitioning or divide and conquer techniques.

2.3.2 Serial Encoding

There are two flavors of serial code structure used in implicitly expressing segmented operations: loop nests and divide and conquer recursion. The primary distinction between the two is that loop nests tend to reflect the nested structure of the representation of the particular data structure being traversed, while divide and conquer recursion tends to reflect the partitioning structure of the algorithm being expressed. In either case the serial code expresses a nesting of control structure.

- Irregular loop nests

26

Adding nesting to recurrent loops usually means one of two things: either the recurrence is carried across the entire loop nest, or there are some completely parallelizable levels of the loop nest. In the former case, the compiler should still try to exploit the recurrence parallelization technique. In the latter case, loop transformation frameworks should be able to expose non-recurrent parallelism in many cases. Loop transformation frameworks are typically constrained to work with perfect loop nests, which are loop nests with code only in the innermost loop. However, there are cases of loop nests which are not fully recurrent for which these frameworks cannot expose any parallelism.

If the loop nest is not a perfect nesting and cannot be made one the compiler may have to compile the recurrence rather than avoid it. Furthermore, if the loop bounds are non-linear in outer loop indices then most loop transformation frameworks are not applicable. We refer to these case as irregular loop nests. The problem with irregular loop nests is that it is difficult to determine the variance or amount of parallelize available at each level of the nesting. Throwing recurrences into the mix makes the task more difficult for the compiler. A recurrence expressed in a loop nest may most naturally and efficiently parallelized using a segmented reduction or scan, depending on the machine target.

- Recursive subroutines

  Algorithms which employ divide and conquer strategies to solve problems typically subdivide the problem into smaller partitions upon which it can recursively and independently work on. That the partitions are worked on independence implies an availability of parallelism between work on the partitions. That the work is done recursive implies a similarity in the type of work done on the partitions that might be amenable to data-parallel style parallelization rather than the more obvious task parallel option.

  With this in mind, the kind of recursive code the compiler targets is that in which each recursive call is independent of its sibling recursive calls. Most divide-and-conquer algorithms on arrays, matrices, or trees fit this particular requirement. (It it is not necessarily true of graph-based algorithms.)

2.3.3 Implementation

The segmentation structure of a segmented reduction or scan can be stripped away (flattened) and a non-segmented scan can be used to perform the computation [15]. The idea is to make the tracking of segment an explicit part of the combining operator. This works well for seg-

mented scans, but there are typically faster mechanism for performing segmented reductions since the only results which need to be propagated back after the initial reduction is for those segments which cross processor/vector register slice boundaries [19].

On distributed memory machines, there are other opportunities for optimizing the implementation of segmented operations. The most obvious is to assure that not segment crosses processor boundaries. However, this may cause load balancing problems since both the partition size and amount of computation per element may vary. Either dynamic load balancing support [44] or algorithmic techniques to ensure balanced computation (i.e. picking good partitioning strategies) are necessary to avoid this problem.

We pursue the flattening strategy in our compiler, since our target is the Cray C90, and architecture which benefits from such an approach.

2.4 Other Primitives

The structure of these recurrent primitives is similar in that operators are used to combine values from a source array (or array expression). What differs is the mechanism by which the source and destination arrays are traversed. In the case of a reduction or scan, the source (and, in the case of a scan, destination) array can be traversed in a fairly linear manner. Combining-send and multiprefix operations traverse the destination array in any order determined by the index vector, while the source array is still traversed linearly.

There are other recurrent primitives that not only traverse arrays differently, but also traverse more complex, recursively-defined structures such as trees and linked-lists. For example, a reduction or scan on a linked list defined with an array traverse both the source and destination arrays in a random order. The following code is an example of such an operation:

```
do i = 1, n
   a(next(i)) = a(next(i-1)) + b(next(i))
end do
```

Of course, if the list is defined recursively using heap-allocated structures or records (as one might using C), then the traversal problem is much more difficult.

Finding associative operators is important in all these recurrent primitives so that reassociation of the operations can be exploited in parallelization. The difficulty of implementing these

other primitives is in developing good parallel templates for performing the operation. For a pointer based structure, this implies a mechanism for supporting pointer dereferences in a globally shared parallel heap.

2.4.1 Implementation

Though we do not compile primitives involving list-ranks, tree reduction, or graph reduction, there is a good deal of work in parallel algorithm development and implementation for these primitives [62][71][77].

2.5 Review

In this chapter, we introduced the basic recurrent primitives we are interested in parallelizing. We also discussed the kind of serial code they occur in and, consequently, we are interested in compiling. Parallel implementations the compiler uses were also reviewed, though, for further detail and insight into their design, we urge the reader to review the source material.

Chapter 3

Recurrent Loops I -
Foundations and Analysis

This chapter presents the intuition and outline of a method for automatically extracting parallel prefix programs and other recurrent primitives from sequential loops. Rather than searching for associative operators in the loop body directly, the method rests on the observation that functional composition itself is associative. We model the loop body as a multivalued function of multiple parameters, and look for a closed-form representation of arbitrary compositions of loop body instances.

This chapter will focus on the basis and motivation for this new analysis. The next chapter will discuss its integration into a compiler and the details of the specific symbolic analysis we use.

3.1 Complexity of Parallel Recurrent Primitives

The parallel reduction and scan operation introduced in the last chapter be used to efficiently execute a wide range of recurrences. For example, the simple recurrence in below, where $\otimes$ is an associative operator, can be solved using a parallel computation of the form in figure 3.1a:

```
do i = 1, n
  a = a ⊗ B[i-1]
end do
```

Figure 3.1b illustrates the more realistic case when $n > p$, where p is the number of processors. A general form of such algorithms can be described simply. The array or expression being reduced

is distributed blockwise across the processing elements. Each processing element performs the reduction locally. Then the results for each processing element are combined in pairwise fashion in a binary combining tree. In the case of a reduction, the algorithm proceeds no further.

The recurrence above can be computed by a parallel reduction. Were this recurrence to compute an array rather than a scalar (i.e. `a[i] = a[i-1] ⊗ B[i-1];`), a parallel scan would be required to solve this so that all intermediate values are computed. The simple algorithm presented above only performs half the necessary work. After the sweep up the combining tree, the partial results in the combining tree are propagated back down to the processing elements, and the local partial sums are updated.

For n array elements and p processors, both parallel reduction and scan operations can be computed on an EREW-PRAM in $O(n/p)$ time steps if $n = \Omega(p\log p)$ [28][53][80] and if the time complexity relationship $T(n) = \lceil n/p \rceil C_1 + C_2 \log p$, such that C_1 and C_2 are both con-
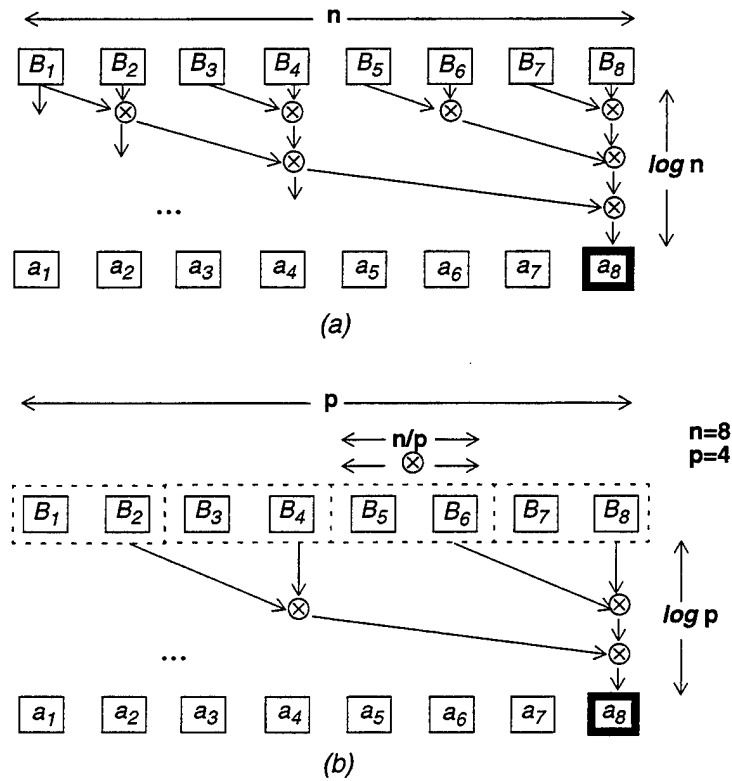
Figure 3.1 An associative recurrence and its parallel combining trees.

stants, holds. The first product term is the cost of the local computation phase of the reduction, and the second product term is the cost of the combining phase of the reduction. For the general recurrence, the time complexity is:

$$T(n) = \sum_{j=1}^{\lceil n/p \rceil} T_{\otimes}(j) + \sum_{i=1}^{\lceil \log p \rceil} (T_{comm}(i) + T_{\otimes}(i))$$

T_{comm} is the cost of communicating the intermediate results of the computation (conceptually, up and down the combining tree). $T_{\otimes}$ is the cost of each application of the associative combining operator. If the operator $\otimes$ were addition, $T_{\otimes} = 1$ and $T_{comm} = 1$, since at most a single number is communicated at a time during the communication phase.

The associativity of the addition operator allows the decomposition of multiply composed additions for parallel execution. Some other examples of associative operators are integer multiplication and MAX. Whether this computation will adhere to a $O(n/p)$ time complexity bound depends on the complexity of each operator application and the cost of communicating intermediate results. It is clear that this will hold if the complexities of both $T_{comm} = 1$ and $T_{\otimes}$ are constant. In the cases of integer multiplication and MAX, it is known that these operations satisfy the criteria necessary to adhere to this time bound.

In serial code, the problem is that we generally have no knowledge of whether a collection of operators in a recurrent loop is associative. So, our goal is to find efficient associative operators in such loops for which we can perform these parallel scans and reductions within the aforementioned time bounds.

3.2 An Associative Model for Recurrent Loop Execution

In our system, recurrent loop bodies are modeled as a series of functions applied to recurrence variables. For example, this loop:

```
do i=1 to n
   a[i] = a[i-1] + B[i-1]
end do
```
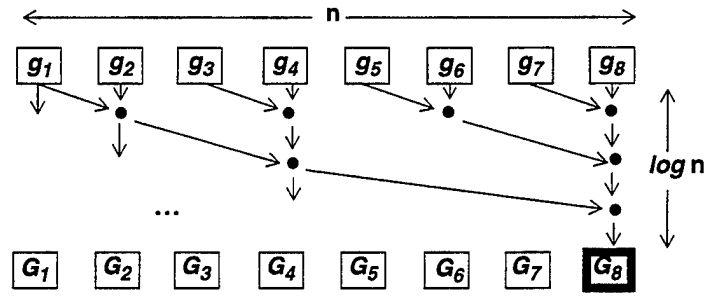
Figure 3.2 Composing the functional model for recurrent loops.

can be modeled by a series of functions[1] $g_i = \lambda x \to x + B_{i-1}$ applied to the recurrence variables a[i], as below:

```
do i=1 to n
   a[i] = gi (a[i-1])
end do
```

We call this function a *loop modeling function*.

The same computation is achieved by precomputing the composed instances of the functions g_i and then applying the function to the first element in the array a, shown below:

```
G1 = g1
do i= 2 to n
   Gi = Gi-1 • gi
end do
do i= 2 to n
   a[i] = Gi (a[1])
end do
```

In this model of the recurrence, we compute final values of the recurrence variable by $a_i = G_i(a_0)$, where $G_i = g_i \bullet g_{i-1} \bullet \ldots \bullet g_1$ and $\bullet$ is the function composition operator over g_i. We refer to the functions G_i as *composite functions*. Note that no dependences hinder parallelization in the application loop. The original loop is effectively "executed" when the com-

---

1. We denote functions using lambda notation because of the ease of manipulating functions in this form.

position operator is applied in a prefix operation over the functions g_i and then each resulting G_i is applied to a_0. The associativity of the composition operator allows for the application of a parallel prefix operation, as in figure 3.2. The case for a composition *reduction* is analogous.

Recall that the time complexity bound of $O(n/p)$ will hold if $n = \Omega(p \log p)$, and T_{comm}, $T_{\bullet}$ are both constant. A naive composition strategy might at least double the complexity of the composed function at each step and yield a result that is as costly to apply as a serial execution of the loop. Furthermore, the cost of communication in the combining steps of a parallel prefix operation is proportional to the size of the functions being communicated. If the function size increases, the communication costs will similarly increase. Under these constraints, finding an efficiently composable function entails finding a composition method with a constant time complexity and a static run-time representation for the modeling function. Requiring that the function representation be closed under composition guarantees both that the representation scheme will be of a fixed size and that the composition method will be of constant complexity.

For example, the composition of two instances of the modeling function for the loop at the beginning of this section is $g_i \bullet g_j = \lambda x \to (x + B_{j-1}) + B_{i-1}$. The composite has doubled in complexity with respect to its two component functions, requiring two additions where the original function had only one. Furthermore, the number of symbolic constants necessary to store with the function has now doubled to two. However, observe that the addition operator in the composed result allows the reassociation $g_i \bullet g_j = \lambda x \to x + (B_{i-1} + B_{j-1})$.

At each composition, the two symbolic constants[2] can be abstracted into one symbolic constant. The resulting composite function $g_i \bullet g_j = \lambda x \to x + C$ has not increased in complexity with respect to the original function. The price paid for this efficiency in representation is that of a single addition operator at each dynamic composition to compute $C = B_{i-1} + B_{j-1}$. Note that this transformation effectively farms out the work to apply the final composite function to the composition method, which will be executed in parallel.

In the dynamic composition phase, it is not necessary to communicate the entire function representation. Only the abstracted constant C is necessary for the composition method we have

---

2. We will generally refer to non-recurrent values as "symbolic constants". Note that also includes literal constants, such as numbers.

derived. In the function application phase, C is used to evaluate the function $\lambda x \to x + C$. Since each composition step takes a constant amount of time and the representation is of a fixed size, the full composition parallel prefix or reduction operation will perform within the $O(n/p)$ time bounds.

3.3 Finding Efficient Composition Operators

The example of the last section demonstrated that a simple symbolic composition scheme can be effective at uncovering efficient composition operators for performing parallel prefix on loop modeling function. The goal of this section is to examine the foundations of such an analysis and provide a rationale and outline of a compiler analysis scheme that we will make more concrete in the next chapter.

3.3.1 Notation and Properties of Modeling Functions

We refer to two expression or functions as *structurally equivalent*, if their expression are isomorphic with respect to all operators and non-bound variables.

The notion of symbolic constants and modeling functions can be viewed as specifying a class of functions. That is, classes of the structurally equivalent functions are defined as follows:

> The class of structurally equivalent functions F is comprised of all functions of the form $f(x_1, ..., x_n, C_1, ..., C_n)$ with equivalent, fixed algebraic structure, where the m bound variables $x_1, ..., x_m$ are fixed and the n variables $C_1, ..., C_n$ vary over values of some specified type.

We require that functions in the class have fixed algebraic structure because all modeling functions look the same, that is, they are structurally equivalent. We will refer more frequently to the *modeling function class*, though the term should be considered interchangeable with *class of structurally equivalent functions*.

Note that individual elements of this class may be uniquely specified by tuples of values for the variables (actually, the symbolic constants in our framework) $(C_1, ..., C_n)$. We refer to each such tuple as a *signature* of the corresponding function, and the space of tuples the *signature-representation* of the function class. To facilitate manipulation of the signature-represen-

tation space, we define an operator to extract signatures from functions of the class F:

$$rep_F(f_i) = (C_1^i, ..., C_n^i).$$

The idea is that the signature-representation of a modeling function class is a more operational representation, since the algebraic structure of the modeling function class is static, and only the symbolic constants are open to manipulation by operators defined reflexively within the class. Conversely, we can define an operator to map a function from signature-representation space to modeling function space:

$$rep_F^{-1}(C_1^i, ..., C_n^i) = f_i.$$

We say that the function class F is closed under composition if there is a composition operator $\bullet$ such that for any $f_i, f_i \in F$, $f_i \bullet f_i \in F$.

Theorem 1 The function class F is closed under composition if there exists a composition operator $\hat{\bullet}$ which manipulates only the signatures $(C_1^i, ..., C_n^i)$ and $(C_1^j, ..., C_n^j)$ for each function f_i and f_i to generate a new tuple $(\dot{C}_1, ..., \dot{C}_n)$ for which the following holds true:

$$rep_F(f_i \bullet f_j) = rep_F(f_i) \hat{\bullet} rep_F(f_j) = (C_1^i, ..., C_n^i) \hat{\bullet} (C_1^j, ..., C_n^j) = (\dot{C}_1, ..., \dot{C}_n).$$

Proof We can easily construct a composition operator of any two functions of F for which it is closed under composition. We first use rep_F to extract their signatures, then use $\hat{\bullet}$ to combine the two signatures, and then to reinject the new signature values into the function structure using rep_F^{-1}. (We are simply constructing the composite operator $rep_F^{-1} \cdot \hat{\bullet} \cdot rep_F$.) By the definition of $\hat{\bullet}$, this is equivalent to $rep_F(f_i \bullet f_j)$. Applying rep_F^{-1} to this gives us $f_i \bullet f_j$, thus constructing $\bullet$ and guarantees that the resulting function is in class F. ∎

We will refer to $\hat{\bullet}$ as a *signature composition operator*.

Given this characterization of function classes, we can now recast our proposed analysis as a search for both a modeling function class F and a composition operator $\hat{\bullet}$ which the corresponding signature-representation class F_{sig} is closed under. In the case of the simple addition reduction example of section 3.2, the composition operator $\hat{\bullet}$ was addition and the signature-representation of the modeling functions were values from the array B.

The analysis starts by extracting a modeling function. The second step of abstracting out constants can be though of as a first cut at defining the modeling function class F. Composing two instances of these functions is equivalent to trying to discover whether this modeling function class is closed under composition. The mechanism by which this was achieved was by extracting out computations solely on symbolic constants (the signature). These computations comprise a candidate composition operator $\hat{\bullet}$.

If the algebraic structure of the resulting composite function is equivalent to that of the original modeling function class, then we need proceed no further. We have a modeling function class which is closed under composition, as well as an operational description in $\hat{\bullet}$ of how to compose the modeling functions while retaining this property. This operator will be that which we will use in our recurrent parallel primitives.

If the algebraic structure of the resulting composite function is not equivalent to that of the original modeling function class, then we can refine our definition the modeling function class by using the composite function as a template for the new modeling function class. The following theorem states that a modeling function class created out of the composition of another modeling function class contains the original modeling function class.

Theorem 2 If we construct a modeling function class F' by composing all possible pairs of instances (i.e. computing the transitive closure over composition) of modeling function class F and $\exists f_{ID} \in F$ such that, $\forall f \in F$, $f = f_{ID} \bullet f = f \bullet f_{ID}$, then $F \subseteq F'$.

Proof For any $f \in F$, $f_{ID} \bullet f \in F'$ by definition of F' and the presumption of the existence of $f_{ID} \in F$. Since $f_{ID} \bullet f = f \bullet f_{ID} = f$, simple substitution gives us $f \in F'$. Thus, $F \subseteq F'$.

∎

Theorem 2 makes the step of redefining our modeling function class intuitive to the extent that we can always generalize the original loops modeling function to the new modeling function class. Note that if $F = F'$, we need not proceed in our analysis. We will have more to say in

section 3.3.4 about whether structuring the search for an appropriate modeling function class in this manner is generally useful.

Note that the Theorem 2 is premised on the existence of identity functions f_{ID}, such that for any F, $f_{ID} \bullet f = f \bullet f_{ID} = f$, $\forall f \in F$. This presents an opportunity to narrow the kinds of modeling functions we can parallelize in this framework by discussing how f_{ID} can be constructed for function classes. (If it cannot be constructed, then our analysis may not be not correct!)

Lemma 1 For modeling functions of linear affine recurrences, f_{ID} can be constructed.

Proof Linear affine recurrences are recurrences have modeling functions of the form $f(x_1, x_2, ..., x_n) = (a_{1,1}x_1 + a_{1,2}x_2 + ... + a_{1,n}x_n + b_1, ..., a_{n,1}x_1 + a_{n,2}x_2 + ... + a_{n,n}x_n + b_n)$. (This is an *nth-order* recurrence.) f_{ID} is constructed by setting $a_{i,i} = 1$ where $1 \leq i \leq n$, $a_{i,j} = 0$ where $1 \leq i \leq n \wedge 1 \leq j \leq n \wedge i \neq j$, and $b_i = 0$ where $1 \leq i \leq n$: $f_{ID}(x_1, x_2, ..., x_n) = (x_1, ..., x_n)$. Simple composition by substitution demonstrates that $f_{ID} \bullet f = f \bullet f_{ID} = f$. ∎

Lemma 2 For modeling functions comprised of conditional expressions and linear affine expressions and inequalities, f_{ID} can be constructed.

Proof We assume, without loss of generality, that the function has no condition expressions nested within relational or affine expressions[3]. Thus, the function can be viewed as a tree of conditional expressions, with affine expressions at the leaves. Some path in the expression tree must be followed down to one of the leaves when evaluating the function. Since the leaves are

---

3. A conditional expression nested within another expression can always be converted to this form by factoring out the conditional expression. For example, the expression $(cnd\ ?TEXP\ :\ FEXP) \otimes EXP =$ can be converted to $(cnd\ ?(TEXP \otimes EXP)\ :\ (FEXP \otimes EXP))$. This operation can be performed recursively until no further conditionals are nested.

themselves linear affine expressions, by Lemma 1 we can construct each to simply evaluate f_{ID} for linear affine recurrences. ■

This raises the question of which modeling function class we should define initially. For some modeling functions, there are potentially infinite variations on the modeling function class that we can choose. We take the conservative approach of minimizing the tuple size for the signatures of the initial function class. For example, given the loop modeling functions (which uses C-style shorthand for conditionals in expressions) $f_i = \lambda x \rightarrow (x < a_i \text{ ? } a_i : x)$, which might be extracted from a MAX reduction loop, there are at least two obvious possibilities for initial modeling function classes. The first possibility is $F = \{f(x, C) = (x < C \text{ ? } C : x)\}$ where $F_{SIG} = \{(C)\}$, for all C. The second possibility is $F = \{f(x, C_1, C_2) = (x < C_1 \text{ ? } C_2 : x)\}$ where $F_{SIG} = \{(C_1, C_2)\}$, for all C_1, C_2.

The second case is more general, including many more functions than are needed initially for this loop. It may turn out that the more general case is closed under composition, while the more restrictive case is not. However, we are better off choosing the more restrictive function class and then generalizing if necessary. We achieve this by abstracting out identical symbolic constants as the same signature constant. Any expressions of the symbolic constants are represented as expressions of the signature constants.

Theorem 3 Given two modeling function classes F and F', such that $F \subseteq F'$ and F' is closed under composition, then for any $f_i, f_j \in F$, $f_i \bullet f_j \in F'$.

Proof The proof is trivial. If $f_i, f_j \in F$, $f_i, f_j \in F'$ since $F \subseteq F'$. Thus, $f_i \bullet f_j \in F'$ since F' is closed under composition. ■

This theorem implies that for any more restrictive modeling function class contained wholly within a more general function class which is closed under composition, then all of its composite functions are contained within the more general modeling function class. The intuition provided by this is that starting with a more restrictive modeling function class is not harmful, in the long run, since the composite functions will always be members of the solution modeling function class, should one exist.

3.3.2 Bounded Recurrences

Callahan [24] coined the term "bounded recurrence" for those recurrences fitting the complexity constraints we presented in section 3.2. A bounded recurrence uses *bounded operators* or *functions,* which are a subset of function classes that are closed under composition. The intuition is to characterize the resource requirements of recurrences which are amenable to parallelization. It turns out that the components of this characterization fit nicely with the model of recurrent loop execution the compiler works with.

The first restriction for a function class to be bounded is that there exists an operator $\hat{\bullet}$ on the modeling function class's signature space F_{SIG} such that the following holds:

$$f_i \bullet f_j \equiv rep_F^{-1}(rep_F(f_i) \mathbin{\hat{\bullet}} rep_F(f_j)).$$

We proved that the modeling function class is closed under composition if such an operator exists in Theorem 1.

The second restriction for *bounded functions* is that the time complexity of rep_F is $O(1)$. What this means, in practical terms, is that the loop modeling functions can be brought into a useful representation in constant time. The third requirement is that the composition function $\hat{\bullet}$ must have an $O(1)$ time-complexity (i.e., the cost of composition is constant and does not grow). The final requirement for a bounded function is that $rep_F^{-1}(rep_F(f_i))(A)$ has a time complexity no greater than $O(f_i)$.

A recurrence whose loop modeling function is a bounded function is parallelizable, since we can find a representation and a composition operator whose complexity is constant. In the example of section 3.2, constructing the composition operator $C = B_{i-1} + B_{j-1}$ in $g_i \mathbin{\hat{\bullet}} g_j = \lambda x.x + C$ satisfies both of these requirements.

The notion of a bounded recurrence is implicit in our model and analysis of recurrent loops. However, we not only seek to verify that a recurrence is bounded by verifying these properties; we also search for a representational function class which both includes the loop modeling functions and is bounded.

f_i | Extract functional model

F | Construct modeling class

$f_i \bullet f_j \in F$ | Compose instances

$F \leftarrow F'$

$\hat{\bullet}, F'$ | Extract symbolic constant terms

$F = F'$? | Test for structural equivalence
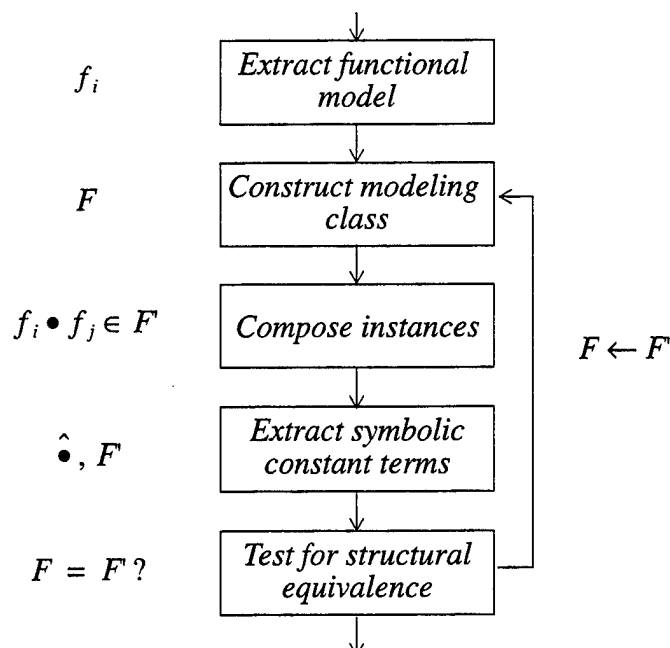
Figure 3.3 Flow diagram for the model and the corresponding model elements computed.

3.3.3 An Analysis Scheme

This simple model is effective in abstracting away syntactic details of a recurrent loop to expose its relevant computational properties as they pertain to parallel recurrent primitives. The example in section 3.2 suggested an automated analysis by which recurrent loop computation is modeled as function composition, outlined in figure 3.3. The analysis will search for composition operators that satisfy our complexity restraints by composing functions using various composition operators and checking whether the loop modeling functions are closed under that operator. The details of this analysis will be discussed in the next chapter. Here, we provide a high level view, referring the more formal notions we have just discussed.

The compiler extracts the loop modeling functions for the recurrent loops by treating the recurrence variables as bound variables in functions. Then symbolic constants and expressions which do no include the recurrence variables are abstracted out as simple symbolic constants. This relieves the analysis of the particular syntax of the code, e.g. details of indexing expressions, etc. It also creates an initial modeling function class F.

Next, the compiler constructs a composition operator. The first step in achieving this is to compose two instances of the modeling function by simple substitution. The second step is to symbolically simplify the composite function by forcing expressions involving only symbolic
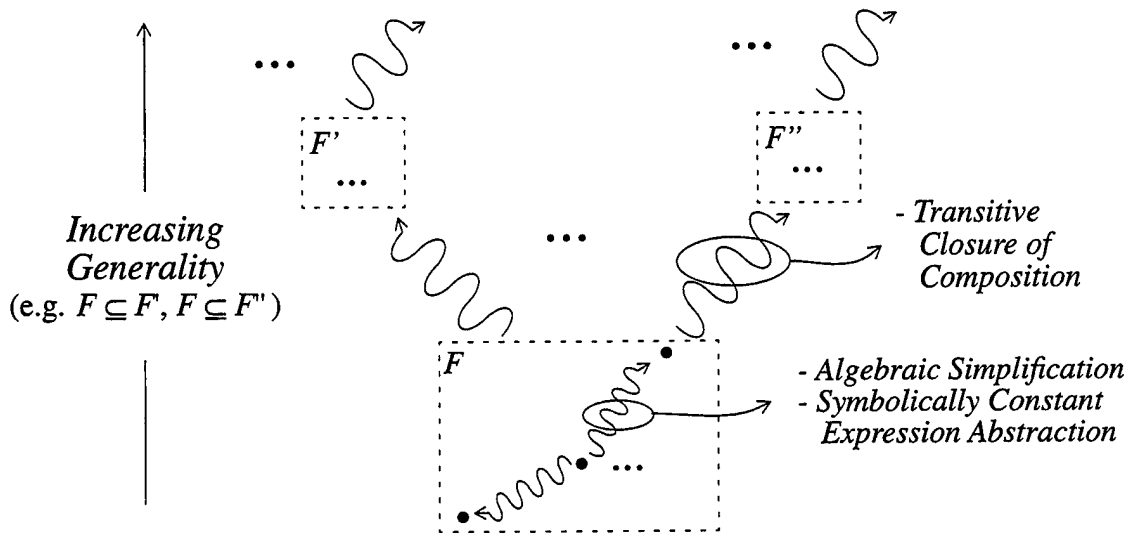
Figure 3.4 Searching for composition operators and loop modeling function classes.

constants to be computed by the composition operator. The net effect of this is to add (constant) complexity to the composition operator, while simplifying the composite function (in the example of section 3.2, extracting the additive expression of the two symbolic constants achieved this). The hope is to find an operator $\hat{\bullet}$ as defined in section 3.3.1. To this end, the goal of the analysis here is to force the composite function to be structurally equivalent to the original modeling functions.

If the analysis does not succeed, it can try again using the (more complex) composite function as the new modeling function. The intuition here is that there may be more general loop modeling functions than our initial choice which are closed under some composition operator. We will discuss the details of this analysis further in the next chapter.

3.3.4 The Search Space

This analysis is essentially a search for a modeling function class and signature composition operator. It is really a search embedded within a search. The first level of search is for an appropriate function class. Within each function class, multiple composition operators may be considered. Figure 3.4 illustrates the search spaces for the analysis. At points in this analysis, we may face may several choices in the search for associative operators. In particular, the

Loop execution
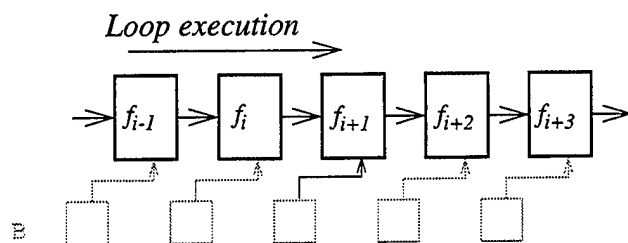
Figure 3.5 Composition of loop modeling functions in reductions and scans.

choices may include deciding how to generalize a (failed) modeling function class, and the ordering of conditionals in expressions when we deal with conditional operators. Backtracking searches or the use of heuristics to guide the search should be used at these points.

If conditional expressions are allowed in the modeling function classes, intractable and undecidable problems will also play a role in the realization of this analysis in a compiler. When simplifying conditional expressions, we will employ logic minimization and linear inequality decision. Also, in general, when testing whether a function class is closed under composition we will test whether a composite function is equivalent to members of that function class, we flirt with the undecidable problem of testing whether piece-wise polynomial expressions are equivalent. In contrast, loops with no conditional constructs (linear recurrences) can be parallelized reliably. We will have more to say about these issues while discussing the implementation of this analysis in the next chapter.

3.4 Modeling Other Recurrent Loops

This modeling scheme is not limited to reductions or scans. Associativity is an important property to discover in many recurrent loops. The model presented here is applicable to such loops. The syntactic similarity of simple reduction or scan loops to *difference equations* [61] made introducing loop modeling functions simpler in that context. But a more general view of loop modeling functions is that they model the computation performed on memory locations. The point of finding associative operators is then to parallelize the computation on each locations. This is clearer when we decouple the notion of collection traversal from the operator used to updates elements of that collection.

The collection traversal scheme that was implicit in the last section was a linear or affine traversal, as in figure 3.5. This illustrates the execution of the loops as a series of applications of the loops modeling function. The source array is accessed sequentially at each loop iteration
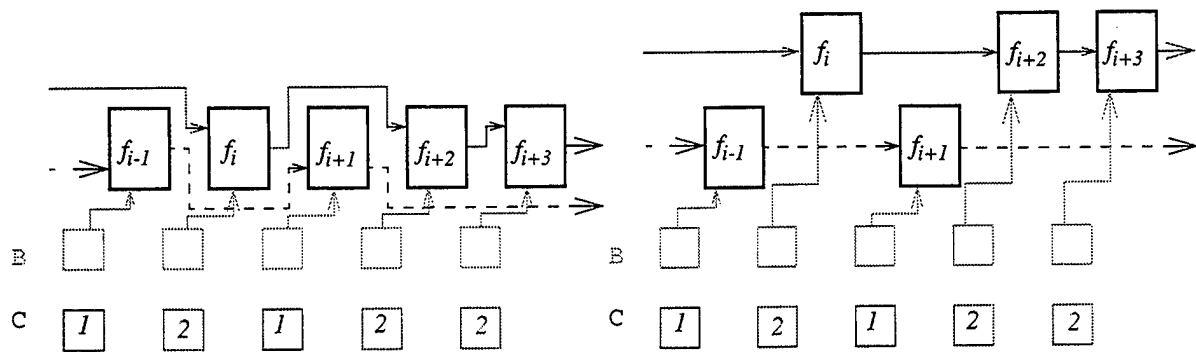
Figure 3.6 Tangled and untangled composition of loop modeling functions in combining-send and multiprefix operations.

and used as an argument to the loop modeling functions f_i. The function composition pattern is fairly straightforward.

A similar illustration of the combining-send or multiprefix operation below is in figure 3.6:

```
do i = 1, n
    ...
    A(C(i)) = fi(A(C(i)),B(i))
    ...
enddo
```

Figure 3.6 separates the operations modifying differing sites in the array A. One can see that there is still a pattern of composing modeling functions in executing this loop. So the model is still valid here. What has really changed is the underlying primitive we will use the associative operator in. The underlying parallel primitive essentially handles the computation and communication patterns, while all the programmer and compiler needs to specify is an associative operator. Thus, the compiler may decouple recurrent primitive determination from the associativity analysis.

This decoupling allows the use of the same associativity analysis for any recurrent code's traversal patterns (for example, the linked list-based reduction or scan in figure 3.7). The overall analysis for recurrent loops will take place in two phases. First, the compiler determines the particular recurrent primitive that is applicable by examining the indexing on the recurrence variables. Then the compiler extracts modeling functions and performs the associativity analysis. These two phases give us all the information we need to parallelize the loop.
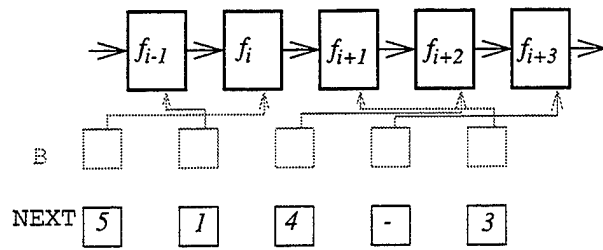
Figure 3.7 Composition of loop modeling functions in list-rank operations.

3.5 Review

This chapter demonstrates that composing instances of the loop modeling functions, simplifying the composite, and abstracting out symbolically constant terms is a promising, sound strategy for automatic parallelization. Chapter 4 discusses the realization of this scheme in our compiler.

Chapter 4

Recurrent Loops II -
Compilation & Code
Generation

This chapter discusses the implementation of the analysis introduced in the last chapter in a parallelizing fortran compiler. The first part of this chapter discusses implementation details of the last chapter's analysis. The second part of this chapter discusses code generation issues relating to parallelized recurrent code.

4.1 Compiler Analysis

The implementation of the associativity analysis is a straightforward adaptation of what was outlined in the last chapter, which we briefly review here. The first step is to extract modeling functions from the recurrent loop. The compiler then starts an iterative process of finding modeling functions closed under composition.

First, the analysis composes two instances of the modeling functions. Next, the analysis simplifies the resulting composite function through algebraic simplification and the construction of a more complex composition operator. Finally, the analysis checks whether the composite is a member of the starting modeling function class. If successful, the compiler can use the constructed composition operator in a recurrent primitive. Otherwise, the analysis continues with the new composite function as the prototypical modeling function. The analysis stops after several iterations through this (the number of iterations can be adjusted, but is fixed at compile time).

Section 4.1.1	*Extract functional model*
Section 4.1.3	*Construct modeling class*
Section 4.1.2	*Compose instances*
Sections 4.1.3, 4.1.4	*Extract symbolic contstant terms*
Section 4.1.5	*Test for structural equivalence*
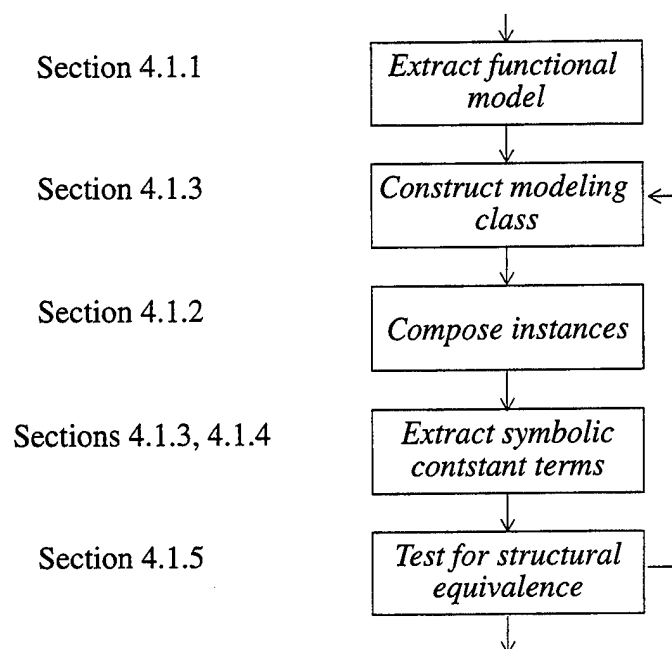
Figure 4.1 Flow diagram for the model and the corresponding model elements computed.

The steps in the analysis are shown in figure 4.1, with cross-references to the sections in this chapter. The sections will refer back to more formal modeling aspects during the discussion of the implementation.

Careful analysis of conditionals allows this analysis to succeed in cases where existing automatic methods fail. Much of the complexity discussed here is due to the introduction of conditional expression, which are more difficult to reason about and manipulate (all of sections 4.1.3.2 and 4.1.4 are concerned with this issue in particular.) The importance of this conditional analysis will become more apparent in later chapters when we deal with more complex control structure and when we examine the problems that pattern matching compilation techniques have with variations of simple recurrences, especially those embedded within conditionals.

4.1.1 Extracting modeling functions

The code in below for computing a linear recurrence (inspired by loop 19 of the Livermore Loops [34]) is used as an example in this section:

```
do i=1,n
   a[i] = E[i] - b[i-1]
```
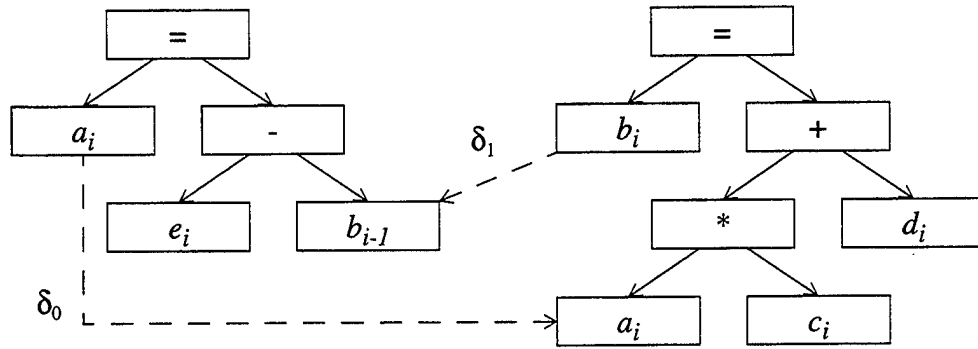
Figure 4.2 A candidate for function modeling in directed acyclic graph form with dependence information.

```
    b[i] = a[i]*C[i] + D[i]
end do
```

The starting point for extracting functional models of loops is extracting strongly connected components of the expression level data dependence graph [93] for the loop body, as in figure 4.2 (only flow dependences are shown). We also include control dependences to account for the presence of conditionals.

Strongly connected components are isolated through loop fission and analyzed as separate recurrences if they have any loop carried flow dependences. In this example, there exists only one strongly connected component. The first pass at extracting functional models assigns a recurrence variable to each flow dependence in the graph. Destination nodes of both inter- and intra-loop flow dependences are replaced by the recurrence variable corresponding to the source of the dependence. The dependence distance determines which instance of the recurrence variable appears. In this code example, we have the coupled recurrences $a_i = E_i - b_{i-1}$ and $b_i = a_i C_i + D_i$.

The next step eliminates non-loop carried dependences to instances of recurrence variables in the expressions. The appropriate recurrence bodies are symbolically substituted for such recurrence variable references within the expression bodies, as in $a_i = E_i - b_{i-1}$ and $b_i = (E_i - b_{i-1})C_i + D_i$.

The loop-carried recurrence variable instances are assigned bound variable names (function parameters). The functions are then constructed by substituting recurrence variable references with the bound variables:

$$f_{a_i} = \lambda(x, y) \rightarrow E_i - y,$$

$$f_{b_i} = \lambda(x, y) \rightarrow (E_i - y)C_i + D_i.$$

It is obvious in this case that this set of recurrences is not mutually recurrent. Only one of the functions need be analyzed and the original forms of the other functions can be used to compute the rest of the recurrence variables' values. The absence of a true mutual recurrence between the functions or subset of functions implies that intra-iteration flow dependences were intrinsic parts of the strongly connected component. These strongly connected components would be broken by the symbolic substitution of intra-iteration recurrence variable references. For example, $f_{b_i} = \lambda x \rightarrow (E_i - x)C_i + D_i$ can be used to generate parallel prefix code to compute `b[i]`. The loop in below is generated to compute `a[i]` and is trivially parallelizable:

```
do i=1,n
   a[i] = E[i] - b[i-1]
end do
```

If the statements were truly mutually recurrent, the functions are combined by forming a function which acts on a tuple. For example, consider the two variable linear recurrence below:

```
do i=1, n
   a[i] = C[i]*a[i-1] + D[i]*b[i-1]
   b[i] = E[i]*b[i-1] + F[i]*a[i-1]
end do
```

The modeling function for this loop is $f_{(a_i, b_i)} = \lambda(x, y) \rightarrow (C_i x + D_i y, E_i x + F_i y)$.

We can also handle *n*th order recurrences such as this:

```
a[i] = f(a[i-1], a[i-2], ..., a[i-n])
```

We transform them into mutual recurrences, as follows:

```
a[i] = f(a[i-1], tmp1, tmp2, ..., tmp(n-1))
tmp(n-1) = tmp(n-2)
...
tmp2 = tmp1
tmp1 = a[i-1]
```

4.1.2 Composing Functions

The easiest step in this analysis is the composition of two functions. The mechanism by which we compose two function is by simple symbolic substitution. Note that this is only during the analysis; at run time, we will compose the functions using the derived operator $\hat{\bullet}$. For example, given two modeling function instances $f_i = \lambda x \to x + C_i$ and $f_j = \lambda x \to x + C_j$, the analysis computes $f_i \bullet f_j = \lambda x \to (x + C_j) + C_i$.

4.1.3 Templatization

In the example presented in section 3.2, we found that, by abstracting out symbolically constant subexpressions, we were able to trade off composition time complexity for composite function complexity. In the addition reduction example, using an addition at composition time fixed the complexity of the composite function[1].

We build the composition operator as we abstract out symbolically constant terms (i.e. simultaneously building both a candidate composition operator $\hat{\bullet}$ and the new function class F'). We call this process of abstraction *templatization*.

4.1.3.1 Polynomial Expressions

Consider this recurrent loop:

```
do i = 1, n
   a[i] = D[i]*a[i-1] + B[i]
end do
```

Its modeling function is $h_i = \lambda x \to D_i x + B_i$. The composition of two abstract instances of h in this initial representation results in:

$$h_i \bullet h_j = \lambda x \to D_i(D_j x + B_j) + B_i = \lambda x.D_i D_j x + D_i B_j + B_i$$

---

1. Another way of looking at why this trade-off is important is that we are transferring complexity from an inherently serial portion to a parallel portion of the computation. The serial portion is that of evaluating each function. The final composite function's computational complexity is a lower bound on the time complexity of its evaluation in the application loop. If some of the computational load is taken up by the composition method, which is applied in parallel, we have effectively converted this time complexity in the form of a serial work load into parallel work complexity.

$$= \lambda x \rightarrow (C_1 x + C_2), \text{ where } C_1 = D_i D_j \text{ and } C_2 = D_i B_j + B_i.$$

No further composition is required as the function class is closed under composition. From this, the composition function is directly inferred as the two operations $C_1 = D_i D_j$ and $C_2 = D_i B_j + B_i$.

We call the functional composite with symbolic constants abstracted out a *template function*. We call the symbolic constants which must be computed dynamically *template variables*.

Template Function[2]:

A template function is a function with symbolically constant subexpressions abstracted out.

Template Variables:

Template variables are the variables used to replace symbolically constant subexpressions which have been abstracted out.

Here, by a simple distribution and an abstraction of coefficients in this polynomial, we have managed to reduce the complexity of the composite function. We have mitigated the complexity of the composite by computing portions of it in the composition operator, rather than deferring evaluation until function application time. We can construct a method by which expressions which are polynomial in the bound variables of a function can be templatized by simply abstracting out symbolically constant coefficient terms as template variables.

The polynomial expression level templatizing algorithm expects that its input is processed so that all multiplications are distributed through additive terms to create a sum of product terms. With this resulting polynomial, the templatizer gathers the coefficients of the bound variables of the function, and then abstracts these coefficients out as template variables, eliminating redundant or equivalent template variables as necessary. This algorithm will always templatize linear recurrences so that the analysis succeeds.

4.1.3.2 Conditional Structure

Conditional structure is more difficult to simplify through algebraic transformation than simple polynomial expressions. For example, we apply a similar line of analysis to a recurrence

---

2. The template function and variables are the components of a modeling function class F.

$$f_i \bullet f_j$$

$B_i > B_j$	$B_j > x$	$B_i > x$	
false	false	false	x
false	false	true	B_i
false	true	false	B_j
false	true	true	B_j
true	false	false	x
true	false	true	B_i
true	true	false	B_i
true	true	true	B_i

(a)

$$f_i \bullet f_j : B_i \leq B_j$$

$B_j > x$	$B_i > x$	
false	false	x
false	true	NP
true	false	B_j
true	true	B_j

$$f_i \bullet f_j : B_i > B_j$$

$B_j > x$	$B_i > x$	
false	false	x
false	true	B_i
true	false	NP
true	true	B_i

(b)

Figure 4.3 Composite functions for maximum reduction problem in switching function form.

with embedded conditionals. Consider the loop body below for computing the maximum element of an array:

```
do i = 1, n
   if (B[i] > max) then
      max = B[i]
   endif
end do
```

The modeling function for this loop is $f_i = \lambda x \to ((B_i > x)?B_i{:}x)^3$. Symbolically composing instances of f results in:

$$f_i \bullet f_j = \lambda x \to ((B_i > ((B_j > x)?B_j{:}x))?B_i{:}((B_j > x)?B_j{:}x))$$

Switching function representations simplify the analysis of conditional nests, as they facilitate the application of standard logic minimization methods. The switching function representation of the composite is shown in figure 4.3a.

The predicate $B_i > B_j$ is symbolically constant. We call symbolically constant predicates *template predicates.*

---

3. modeling functions are extended to include a conditional operator and relational operators. We use the conditional notation *(predicate? trueval: falseval)* for conciseness.

Template Predicate:

A template predicate is a relational expression in which all embedded terms are symbolically constant.

The template predicate is abstracted out and evaluated during each dynamic composition, yielding the two switching functions in figure 4.3b. Note that certain table entries here require logical contradictions between the predicates to be selected. These entries are replaced with 'NP' in the entries above.

Applying the logic minimization method we discuss in section 4.1.4 will yield:

$$f_i \bullet f_j = \begin{pmatrix} \lambda x \to (B_i > x)?B_i:x & \text{if } (B_i > B_j) \\ \lambda x \to (B_j > x)?B_j:x & \text{if } (B_i \le B_j) \end{pmatrix}$$

While these two functions seem to indicate that the original modeling function was closed under composition, somehow they must be unified into one representation while retaining this desirable property. Note the similarity in the structure of the two simplified functions. They are nearly identical, except where symbolic constants occur. We call this condition *structural isomorphism*.

Structural Isomorphism:

Two expressions are structurally equivalent if, when abstracting out symbolically constant subexpressions, there exists some ordering of the predicates in the conditional nest for which the CNF-Exp representing them is isomorphic.

Figure 4.4 graphically depicts these two functions. We can unify these two functions by creating an isomorphism mapping between them. The composite functions in this example are unified by creating the mapping $((B_i, B_j))$. The template variables are selected from the mapping(s) based on the template predicate: $f_i \bullet f_j = \lambda x \to (C > x)?C:x$, where $C = (B_i > B_j)?B_i:B_j$, as figure 4.4 illustrates. This template function is structurally equivalent to the original looping modeling function, confirming closure under the newly constructed composition operator.

At the expense of one comparison operator at each dynamic composition, we have guaranteed that the resulting composite function does not increase in complexity. The effective run-time representation of intermediate composite functions is the template variable C, which we use
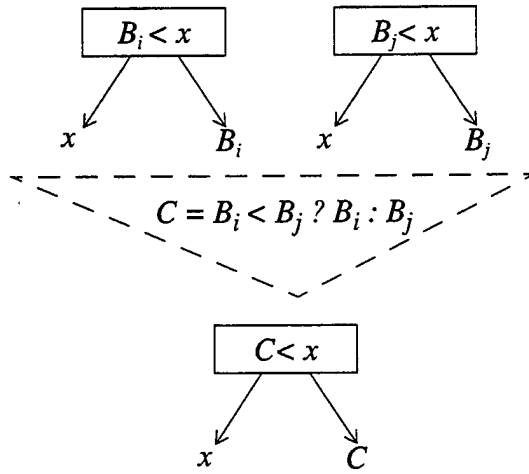
Figure 4.4 Unifying subfunctions with
a template predicate.
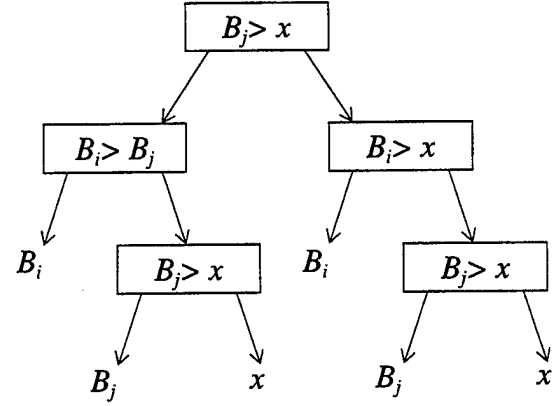
Figure 4.4 The composite maximum
function in CNF-Exp form.

to evaluate the template function in the application phase. Thus, the solution conforms to the $O(n/p)$ time bounds.

While analyzing this problem, we took advantage of the fact that predicates with template predicates may be evaluated at composition time. Other than the conditional optimizations we discuss in section 4.1.4, the only way to reduce the complexity of composed conditional nests is to abstract out template predicates and add them to the composition operator. However, when template predicates are abstracted out, multiple versions of the function are generated for each combination of logical values of the template predicates. Unifying these subfunctions to reduce the complexity of such a composite implies finding some structural correlation between them. An simple way to find a correlation is to check for structural equivalence in the conditional nest structure. If the isomorphism exists, we find polynomial expression templatizations in the leaves of the conditional nest for each subfunction and then pick among them by evaluating template predicates at composition time. It is by this unification mechanism that we reconcile the simple polynomial expression templatization algorithm with template predicate extraction in conditional nests.

To facilitate the transformations and traversals necessary for this algorithm, we convert conditional nests to a normal form, called *Conditional Normal Form Expression* (CNF-Exp).

Condition Normal Form Expression:

A CNF-Exp is defined to be conditional nest in which all conditional expressions are distributed out of arithmetic operators, relational operators, and the predicates of conditional operators.

Figure 4.4 illustrates a CNF-Exp form for the first composite of the maximum reduction example. We normalize functions into CNF-Exp form after logic minimization and template predicate abstraction. This allows for easy structure traversal when checking for isomorphism.

If an isomorphism mapping cannot be found to unify subfunctions, then the analysis terminates with failure. Detection of structural isomorphism and extraction of mappings is a potentially costly operation, given that the number of possible condition orderings in a conditional nest are exponential. However, in the minimization process we use some heuristics to order the conditions so that we can apply a simple linear walk over the expressions to construct the mapping and detect structural isomorphism. The ordering heuristics checks only that the conditions of the same rank in the conditional nests of subfunctions are themselves structurally equivalent to each other. The algorithm for templatization is described in high level applicative style is below. The algorithm assumes that its inputs are a set of functions in CNF-Exp form and preprocessed for conditional ordering:

```
function SimpleTemplatize(SoPs, TempVarSet)
      ;; collect polynomial coefficients
   foreach (ProdTerm in SoPs)
      Coeff = GetCoefficients(ProdTerm)
         ;; check for redundancy
      if (not (newTempVar == find(Coeff, TempVarSet))) then
         newTempVar = GenTempVar()
      end if
      add(GenTempVar(), Coeff, TempVarSet)
      replace(Coeff, newTempVar, SoP)
   end foreach
   return TempVarSet
end

function Templatize(CNF-Exprs, CurrTemp)
   if (PolyExprs(CNF-Exprs)) then
         ;; polynomial leaf in all subfunctions
      if (not (SimpleExprIsomorph(CNF-Exprs)) then
            ;; no isomorphism between subfunctions
         exit(Failure)
      end if
      return SimpleTemplatize(CNF-Exprs, CurrTemp)
```

```
else if (CondExprs(CNF-Exprs)) then
        ;; conditional node in all subfunctions
    return Templatize(map(RightExpr, CNF-Exprs),
                Templatize(map(LeftExpr, CNF-Exprs),
                    Templatize(map(PredExpr,CNF-Exprs),
                                    CurrTemp)))
else
        ;; no isomorphism between subfunctions
    exit(Failure)
    end if
end
```

There is a complication that the maximum reduction example does not expose. The simplified composite function

$$f_i \bullet f_j = \begin{pmatrix} \lambda x \to (x + B_j + B_i < 0)?0:x + B_j + B_i \text{ if } B_i < 0 \\ \lambda x \to (x + B_j < 0)?B_i:x + B_j + B_i \text{ if } B_i \geq 0 \end{pmatrix}$$

is from the example in section 4.2. A naive templatization from the mapping $\{(B_j + B_i, B_j),$ $(0, B_i), (B_j + B_i, B_j + B_i)\}$ might produce three template variables for the template function $\lambda x \to (x + C_1)?C_2:x + C_3$, where $C_1 = (B_i < 0)?(B_j + B_i):B_j$, $C_2 = (B_i < 0)?0:B_i$, and $C_3 = B_j + B_i$.

However, the relationship $C_1 + C_2 = C_3$ suggests an alternative abstraction $\lambda x \to (x + C_1)?C_2:x + C_1 + C_2$, with the same values for C_1 and C_2. This relationship might be important in subsequent compositions of the composite function. We avoid this over-abstraction by eliminating those template variables which can be expressed as simple linear combinations of other template variables.

4.1.4 Simplification of conditional operators

Simplification of conditional operators is essential for dealing with recurrences containing conditional control structures. The basic goal of the conditional analysis phase is to find and eliminate logical discrepancies and redundancy within an expression.

The form of simplification shown in figure 4.5a involves redundancy; if a conditional node has identical children, the conditional node can be eliminated. Another involves a discrepancy in which identical predicates are in an ancestor/descendant relationship within an expression, as in figure 4.5b. The first expression represents such a situation. As the conditional node with
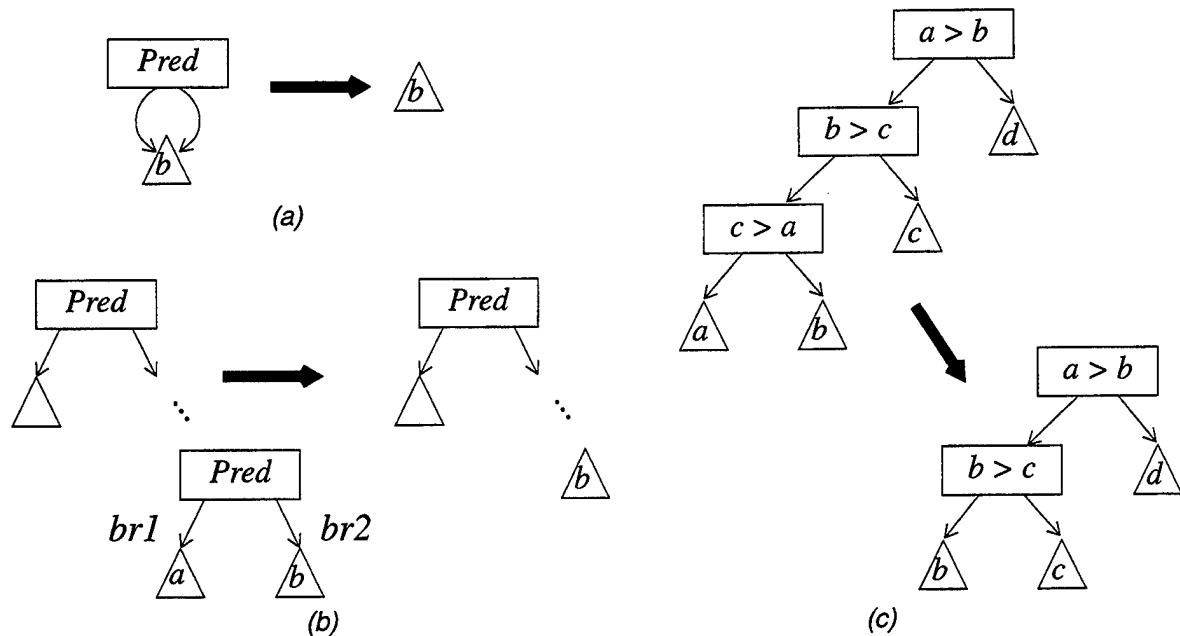
Figure 4.5 Simplifications of conditional nests by exploiting redundant and infeasible paths.

the identical predicate is a descendant of the false branch of the root condition node, the true branch of that node will never be evaluated, allowing the elimination of that branch. In other words, the conditional subnode becomes unnecessary as its evaluation is predetermined by its status as a descendant of a conditional node with an identical predicate. The second expression is the result of this simplification.

Other logical discrepancies may require complex symbolic reasoning to deduce relationships between dissimilar predicates; for example, the predicates $a < b$, $b < c$, and $c < a$ can never all evaluate as true, allowing the simplification in figure 4.5c. The module which uncovers logical discrepancies between the potential logical evaluations of the predicates in the conditional nest can be structured in several different ways. As the analyses involve linear relational expressions, linear or integer programming formulations of the problem are options. However, the complexities of such algorithms (integer linear programming is NP-complete) make other approaches worth considering. We resort to a pruning, heuristic search of the exhaustive set of subsets of the predicates in a conditional nest. In spirit, it is an optimized version of the Fourier-Motzkin method for deciding linear inequalities [33]. Other alternatives include fast methods by Shostak for handling special forms of linear inequalities [78].

Optimizations of conditionals are based on the switching function representation introduced earlier. The switching function is created by tracing paths from the root to the leaves in the conditional nest, forming table entries. Note that we do not build an exhaustive switching function representation, though, for conceptual purposes, they are represented as such in this paper. Simple conditional contradictions of the form in figures 4.5a and 4.5b are eliminated in the tabulation process. Logical inconsistencies are eliminated by striking lines from the switching function. Template predicate extraction is trivially achieved by generating multiple sub-tables of the original switching function for each possible evaluation of the template predicate. Conditional redundancies are eliminated by applying multilevel logic minimization to the representative switching function. We base this step on the standard logic minimization package, Espresso [23].

Consider the composite function (in CNF-Exp form) from the example in figure 4.4:

$$f_i \bullet f_j = \lambda x \rightarrow (((B_j > x)?(B_i > B_j):(B_i > x))?B_i:((B_j > x)?B_j:x))$$

This function was first converted to the switching function form, as demonstrated in section 4.1.3.2. Note that while the first occurrence of the predicate $B_j > x$ seems to indicate that there may be a circumstance under which this predicate is true and the value x would be returned by the function, the second occurrence predicate guarantees that this will never happen. The switching function representation indicates this. This is an example of the tabulation eliminating a simple conditional contradiction. The subfunction switching functions generated by removing the constant conditional $B_i > B_j$ are trivially derived from the switching function. The conditional contradictions arising from conflicting predicates are easily struck from these switching functions. Finally, the redundant conditional nodes were removed by the logic minimization procedure.

4.1.5 Testing for Closure

Testing whether two modeling function classes are closed under composition is fairly easy in this framework. The template functions and template variables are those entities that we will compare. Since we use ordering heuristics, discussed in prior sections, to generate the template functions, then comparison need only be syntactic. The syntactic check for structural equivalence between two template functions, which runs in time linear in the size of the functions, is defined recursively below in pseudocode:

```
function StructuralEquivalence (E1, E2)
   case type(E1) of
      SymbolicConstant:
         return (type(E2) == SymbolicConstant);
      BoundVariable:
         return (type(E2) == BoundVariable);
      ConditionalExpr:
         return (type(E2) == ConditionalExpr)
                  and StructuralEquivalence(PredExpr(E1),
                                                  PredExpr(E2))
                  and StructuralEquivalence(LeftExpr(E1),
                                                  LeftExpr(E2))
                  and StructuralEquivalence(RightExpr(E1),
                                                  RightExpr(E2)));

      Expression:
         return (type(E2) == Expression)
                  and (operator(E1) == operator(E2))
                  and StructuralEquivalence(LeftExpr(E1),
                                                  LeftExpr(E2))
                  and StructuralEquivalence(RightExpr(E1),
                                                  RightExpr(E2)));

   end case
end
```

The final check is to assure that there are the same number of template variables in each function class. This assures that both signature spaces are equivalent.

4.1.6 Scope

The framework presented here is general, however, the specific techniques are tune to a particular class of functions. This function class is defined simply as the class of n-dimensional piece-wise linear functions with symbolic constants. We define piece-wise linear functions to be linear expressions embedded within relational expressions and conditional operators.

This technique can analyze the class of polyregion-wise polynomial functions. Polyregion-wise polynomial functions are the complete class of functions defined over multiplication, addition (and subtraction), and conditional and relational operators. However, the underlying abstraction mechanism and analytical techniques are suited more toward piece-wise linear functions.

Note that we do not guarantee finding solutions if they exist. Some of the techniques presented here are necessarily heuristic in nature. Implementing a full search through the space of functions is intractable, both theoretically and practically, as some of the subproblems presented

60

here are either intractable or undecidable in nature. From a compiler perspective, trading off some thoroughness for speed is necessary. However, standard techniques for dealing with these problems efficiently have been utilized in other domains. In the case of logic minimization, work in VLSI design has been a particularly rich source of efficient algorithms [23]. The problem of deciding linear inequalities has occurred elsewhere in compiler work [69][86], as well as a multitude of other research domains.

4.2 Example: Maximum Subsequence Sum

This code in below computes the largest non-negative contiguous subsequence sum of a series of real numbers.

```
do i = 1, n
   if (sofar + B[i] < 0) then
      sofar = 0
   else
      sofar = sofar + B[i]
   end if
   if (max < sofar) then
      max = sofar
   end if
end do
```

The second statement in this loop body is simply an articulated maximum reduction. For the purposes of this demonstration, the variable `sofar` is promoted to a vector so that the loop may be split between the two conditional statements.

The loop modeling function is $f_i = \lambda x \rightarrow (x + B_i < 0)?0:x + B_i$. The first composition results in the following function:

$$f_i \bullet f_j = \lambda x \rightarrow (((x + B_j < 0)?0:x + B_j) + B_i < 0)?$$

$$0:((x + B_j < 0)?0:x + B_j) + B_i .$$

The switching function representation of this function is shown in figure 4.6b. Extracting the template predicate $B_i < 0$ gives the switching functions in figure 4.6c. The three predicates ($B_i < 0$, $x + B_j < 0$, $x + B_j + B_i < 0$) contradict with the logical evaluations (true, true, false) and (false, false, true). These cases were struck from the switching functions.

61

$$f_i \bullet f_j$$

$B_i < 0$	$x + B_j < 0$	$x + B_j + B_i < 0$	
false	false	false	$x + B_j + B_i$
false	false	true	0
false	true	false	B_i
false	true	true	B_i
true	false	false	$x + B_j + B_i$
true	false	true	0
true	true	false	0
true	true	true	0

(b)

$$f_i \bullet f_j : B_i < 0$$

$x + B_j < 0$	$x + B_j + B_i < 0$	
false	false	$x + B_j + B_i$
false	true	0
true	false	NP
true	true	0

$$f_i \bullet f_j : B_i \geq 0$$

$x + B_j < 0$	$x + B_j + B_i < 0$	
false	false	$x + B_j + B_i$
false	true	NP
true	false	B_i
true	true	B_i

(c)

Figure 4.6 Switching function representations for the maximum subsequence problem.

Logic minimization gives the following functions:

$$f_i \bullet f_j = \begin{pmatrix} \lambda x \to (x + B_j + B_i < 0)?0 : x + B_j + B_i & \text{if} & B_i < 0 \\ \lambda x \to (x + B_j < 0)?B_i : x + B_j + B_i & \text{if} & B_i \geq 0 \end{pmatrix}.$$

The isomorphism mapping for these two functions is $\{(B_j + B_i, B_j), \quad (0, B_i),$ $(B_j + B_i, B_j + B_i)\}$. There are two alternatives in templatization. Over-abstracting would eliminate the quantitative relationship between the first two pairs and the third, namely, that the third pair is the sum of the first two.

The resulting templatized composition is:

$$(f_i \bullet f_j) = \lambda x \to (x + C_1 < 0)?C_2 : x + C_1 + C_2,$$

$$\text{where } C_1 = (B_i < 0)?B_j + B_i : B_j$$

$$\text{and } C_2 = (B_i < 0)?0 : B_i.$$

As this is not structurally equivalent to the original function, we must recompose with the new template function $f_i = \lambda x \to (x + C_{1i} < 0)?C_{2i} : x + C_{1i} + C_{2i}$.

The templatized recomposition of this function will reveal that this template function is closed under composition, yielding the following scheme for combining template variables:

$$C_1 = ((C_{1i} + C_{2j} < 0)?C_{1i}:C_{1j} + C_{1i} + C_{2i})$$

$$C_2 = ((C_{1i} + C_{2j} < 0)?(C_{1j} + C_{2j} + C_{2i}):C_{2j})$$

4.3 Code Generation

4.3.1 Code Templates

The compiler now uses the efficient composing operator it has derived to compute the recurrence in parallel. The first step is to compute the template variables described earlier in this chapter. These form the initial signatures used for each modeling function. Then the signatures are composed using a parallel reduction or scan on the composition operator. Finally, the resulting signatures for the composite functions are used to compute the final recurrence values. The primitives used are architecture dependent. On the Cray C90, we use those described in chapter 2.

The basic template the compiler uses is illustrated in below, using the formal concepts we discussed in the last chapter:

1. Compute rep_F for the loop functions

2. $sig._{\hat{\bullet}} = \text{parallel\_compose}(\hat{\bullet}, rep_F)$

3. Compute $(rep_F^{-1}(sig._{\hat{\bullet}}))(\text{initial vals})$

The following pseudo-code is closer to the compiler target and uses the terminology of this chapter:

```
<original loop header>
   initialize template variables
<end loop>
new_template_vars = recurrent_primitive( •̂ , <template vars>)
<original loop header>
   use new_template_vars in template functions
      and apply to initial value of
         recurrence variables
<end loop>
```

For example, for the simple linear recurrence of section 4.1.3.1, this piece of code looks like the following:

```
do i = 1, n
    tmp1[i] = D[i]
    tmp2[i] = B[i]
end do
```

$$(\texttt{newtmp1[1:n]},\texttt{newtmp[1:n]}) =$$
$$\texttt{scan}([(tmp1, tmp2) \oplus (tmp1', tmp2') \rightarrow (tmp1 \cdot tmp1', tmp1 \cdot tmp2' + tmp2)],$$
$$(\texttt{tmp1[1:n]},\texttt{tmp2[1:n]}))$$

```
do i = 1, n
    a[i] = newtmp1[i]*a[0] + newtmp1[i]
end do
```

The first step is to initialize the template variables for the modeling function classes. These are used in the next step, which is to compute the composition of modeling functions using the derived composition operator and the template variables. Using the resulting template variables, the values of the recurrent variables are computed using the initial value(s). Both loops and the primitive here are parallelizable.

4.3.2 Primitive Selection

Deciding which primitive to use depends on the form of the indexing patterns on the recurrence variables. Examples of the patterns of primitives we compile are listed below:

- Reductions

  ```
  do i = 1, n
      a = f(a)
  enddo
  ```

- Scans

  ```
  do i = 1, n
      a(i) = f(a(i-1))
  enddo
  ```

- Combining-sends/Multiprefix

  ```
  do i = 1, n
      a(c(i)) = f(a(c(i)))
  enddo
  ```

These are only one of many potential encodings for each primitive. The compiler does not rely solely on indexing expression. Rather, it examines the loop carried flow dependences in recurrent cycles and their dependence distances to determine the type and order of the recurrence. This simple algorithm abstracts away the details of the indexing expressions and syntax, relying primarily on the abstract representation of the dependence graph.

The presence of a maximum loop-carried dependence distance of two would indicate a second-order recurrence computable by a reduction or scan, and so on. When there is no discernible dependence distance, the compiler can check to see whether or not the indexing expressions are identical to each other, in which case it can generate code for a combining-send or multiprefix operation, depending on whether intermediate values are used.

In cases of complex indexing which do not satisfy these requirements, run-time tests may be employed to check whether a particular index set has properties similar to any of these primitives. This would probably only be worthwhile if the index sets are reused, so that the cost of the run-time tests are mitigated.

4.4 Review

The observations that function composition is associative and that recurrent loop bodies can be modeled as function applications lead us to a framework for extracting reduction and parallel prefix operations in a general and flexible way. The compiler now has a powerful and flexible kernel for parallelizing recurrent code, upon which it can now layer control structure transformations to handle nested of computation. The control structure transformation we will describe in subsequent chapters will rely heavily on the ability of the compiler to deal with conditional expressions.

Chapter 5

Irregular Control Structure I - Loop Flattening

Irregular loop nests in which the loop bounds are determined dynamically by indexed arrays are difficult to compile into expressive parallel constructs, such as segmented scans and reductions. In this chapter, we describe a suite of transformations of irregular loop nests to enable automatic parallelization by both tradition parallelization techniques and recurrence analysis. The basis is a simple, general loop flattening transformation, along with new optimizations which make it a viable compiler transformation. Coupled with the recurrence parallelization technique of the last two chapters, the transformation enables parallelization of segmented reductions and scans.

5.1 Irregular Loop Nests

Many real world parallel applications display little regularity in their computational structure. Often, communication and computation patterns are determined dynamically from data values. A typical example is the use of sparse matrix and vector representations. Most attempt to conserve memory and computation by storing only computationally relevant portions of the data (i.e. non-zeros) and a representation space mapping. Other applications may require that a data structure be partitioned in an irregular fashion. In some cases, this gives rise to code in which control structure depends on dynamically determined data values, such as partition points or pointers into index space. Applications which may exhibit these properties include sort, text processing, computational geometry, image processing, and molecular dynamics simulations.

High level parallel primitives, such as reduction and scan, merge, and set operations, are useful in parallelizing many of these *irregular* computations. These primitives can display markedly better

speedup and/or load balancing characteristics than alternatives. It is important in these cases that a parallelizing compiler have the ability to infer such high level primitives from complex control structure.

Consider the sparse matrix-vector multiplication loop below:

```
do i = 1, N, 1
   y(i) = 0.0
   do k = pntr(i), pntr(i+1)-1, 1
      y(i) = y(i) + val(k)*vec(indx(k))
   end do
end do
```

This loop utilizes the popular Compressed Sparse Row (CSR) representation of sparse matrices [32], which is also illustrated below.

```
VAL:    [1 2  3  4  5  6  7  8  9  10 11]
INDX:   [3 5  4  5  1  3  4  2  5  1  6]
PNTR:   [1    3     5  6     8     10    12]
```

CSR representation consists of an array of nonzero values of the matrix in row-wise order (val), an array of corresponding column numbers (indx), and an array pointers to the beginning of each row (pntr). The inner loop bounds and trip count are determined solely by the array pntr, whose value is generally not known to the compiler. Consequently, despite potential benefits (i.e. load balancing, availability of parallelism) in parallelizing across all inner loop iterations, it is difficult to compile this way.

Successful attempts at manual parallelization of the CSR kernel have utilized segmented reductions [20] (many irregular computations can be effectively parallelized using segmented reductions and scans [15]). Segmented operations allow for arbitrary nesting structure to be imposed on the source array. The operation is evaluated independently in each segment, resetting at the beginning of each segment, as in the example of Figure **. Explicit segmented reduction, scan, and elementwise operations can be flattened so that they maximize available parallelism in the operation and simplify the load balancing problem. Our compilation approach automatically detects these segmented operations and flattens them.

Combining-send and multiprefix operations have also been shown to effectively parallelize sparse matrix kernels [77]. The second representation and kernel below are called Compressed Sparse Column (CSC):

```
VAL:    [5 10 8  1  6  3  7  2  4  9  11]
INDX:   [3  6  5  1  4  2  4  1  2  5  6]
PNTR:   [1    3  4    6    8       11 12]

do i = 1, N, 1
   do k = pntr(i), pntr(i+1)-1, 1
      y(indx(k)) = y(indx(k)) + val(k)*vec(i)
   end do
end do
```

The CSC representation is essentially equivalent to the CSR representation of the transpose of the matrix. Current compilers might parallelize the inner loop, generating a simple combine-send of limited parallelism nested within the outer loop by using pattern matching or some other ad-hoc technique. However, a better approach might be to parallelize the entire loop nest using a single combining-send with a more complicated operator. What the compiler will derive is not a segmented operation, in this case, but a combining-send operation to compute the entire loop.

We refer to loop nests which display the type of loop bound indirection evident in this example as *irregular loop nests*, as opposed to *regular loop nests* in which the loop bounds are "analytically manageable" by the compiler. One might consider irregular any loop nest whose closed form cannot be determined by the compiler. With current analytical skills of compilers, this may be any loop bound that is not either constant or a linear combination of outer loop indices. The boundary between regular and irregular loop nests may change with the sophistication of the compiler. However, the class of loops we are interested in derive their iteration space shape from dynamically determined data values about whose patterns the compiler is likely to know nothing, so we are solving a more general problem. Other types of irregular loop nests may involve nested while loops which have similar indirection in the loop exit conditions.

Our goal in automatically parallelizing these irregular, recurrent loop nests is to automatically construct the high level parallel primitives described above from serial code in a reasonably general manner. The parallelization technique presented in previous chapters can parallelize recurrence well in the presence of complex conditional statements. This enables parallelization of explicitly segmented computations (in non-nested loops). However, many segmented

operations are expressed implicitly through loop nests. We seek to transform these loop nests so that our recurrence parallelization technique and other standard parallelization transformations can be applied across the entire loop nest.

We use a basic loop flattening transformation that facilitates the parallelization of irregular loop nests and provides a basis for recognizing more sophisticated parallel primitives. The basic idea is to create a single, non-nested loop that emulates the execution of the original loop nest. This is achieved by first computing the original loop nest's index sets. Then, by creating a non-nested loop with a trip count equal to the total sum of inner loop trip counts, the precomputed index sets are used to decide which point in the original loop the flattened loop should execute. The transformation has the following benefits:

- The application of most existing parallelization transformations as well as our recurrence parallelizing technique to the loop nest *in toto* by applying the transformations to the *flattened* loop. Artifacts of the loop flattening transformation include complicated conditional structures, which the recurrence parallelization technique must be able to deal with.

- The parallelization of the index set computation. This allows the practical use of loop flattening in a compiler by attacking the remaining artifact of the transformation: the index set computation. Also, this leads to some intriguing future possibilities for extracting other sophisticated algorithmic idioms from irregular loops.

- The amortization of the index set computation over repeated executions of the loop nest. For sparse matrix-vector multiplication, this is analogous to the preprocessing steps of many existing parallel libraries, most of which are applied once for repeat multiplications of the matrix.

Applying parallelizing transformation *in toto* is important for the following reasons:

- Load balancing the assignment of outer loop iterations is complicated by unpredictable inner loop trip counts.

- Inner loop trip counts may not be sufficiently large to make parallelizing the inner loop body worthwhile.

This paper will focus on a particular type of irregular loop nest, which we refer to as a *segmented loop nest*, in which the inner loop bounds are functions of array references indexed by loop index variables and are invariant with respect to the loop nest. However, since the overall
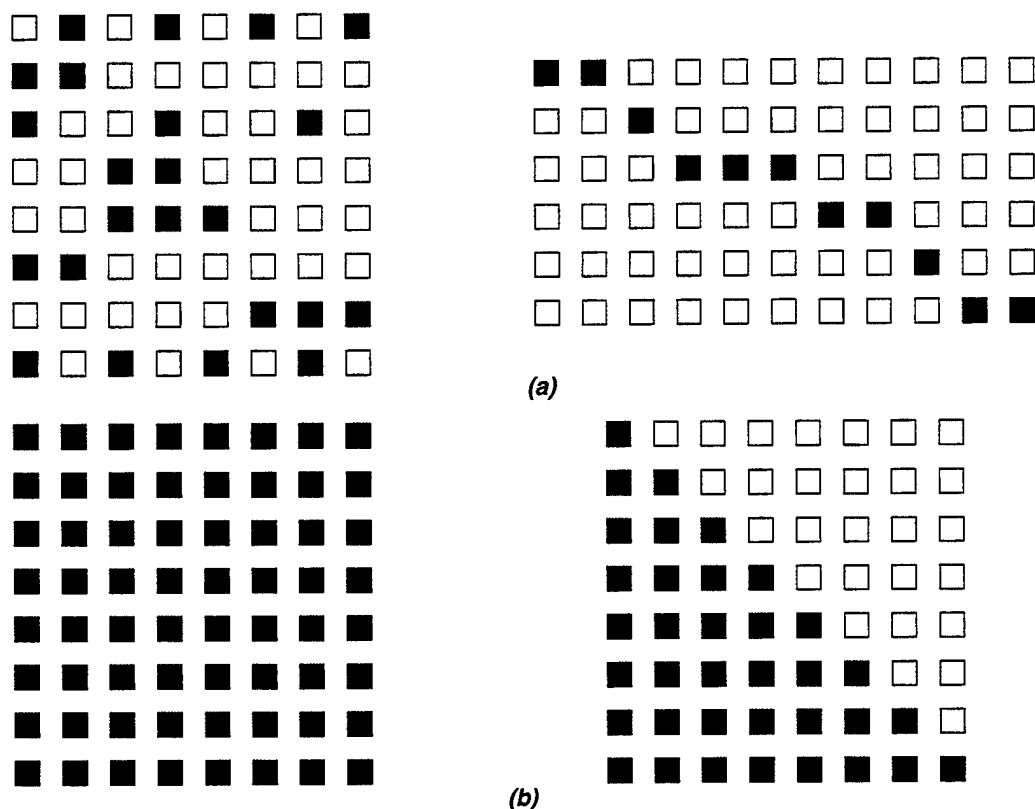
Figure 5.1 Some possible index space iterations by (a) irregularly nested loops and (b) regularly nested loops.

framework is general, this may be a viable approach for more complicated irregular loop nests, such as those that must be parallelized using parallel merge and pattern matching.

5.2 Parallelization Strategies

The shape of the index space traversal of an irregular loop nest may vary arbitrarily, as in Figure 3a. A nested while loop without a discernible induction variable, or with an induction variable with a dynamically determined bound, may also be considered an irregular loop nest. Some common index space traversal patterns that can be considered regular are shown in Figure 3b.

As previously mentioned, irregular loop nests occur in many applications, including sorting, computational geometry, molecular dynamics, image processing, and sparse algorithms. We will focus on the examples given in the introduction involving sparse matrices, although the majority of the discussion and optimizations here apply to all these algorithms.

Consider again them CSR sparse matrix-vector multiplication loop nest, in Figure 1. There are a variety of strategies for parallelizing this:

- *Parallelize across the outer loop iterations.* This approach would assign outer loop iterations to processors. The problem here is that the inner loop trip counts may be computed at run time and may vary arbitrarily, making static load balancing impossible for this approach. This approach can be easily automated, but is not always possible for the other irregular loop nests we have discussed, such as the CSC kernel of Figure 1.

- *Parallelize the inner loop.* This approach would attempt to parallelize the inner loop only (in this case, a reduction). The problem here is that the inner loop trip counts may be small. In the case of a sparse matrix, the average sparsity of the rows would probably not yield sufficiently long running inner loops. Rudimentary recurrence parallelization capability makes this viable approach for an automatically parallelizing compiler.

- *Pad the inner loop trip count.* This approach would use the maximum inner loop trip count as the trip count for all the outer loop iterations. The main problem here is that useless computation is performed for the sake of regularity in the loop nest. This approach is less efficient than one which can perform a global parallel operation to compute the entire loop nest without padding. Furthermore, the data structure for the sparse matrix will waste memory. This is the approach taken by Ellpack/Itpack storage method [51]. This approach would be difficult, and not particularly worthwhile, to generally automate.

- *Parallelize across the entire loop nest with a segmented reduction operation.* This solution, proposed by Blelloch et al. [20], does not pad the computation. Instead, it treats the values of the CSR representation as a *segmented vector*, with each row treated as a segment. The addition portion of all the row-vector inner products can be performed using a single segmented reduction. The problem of load balancing reduction and scan operations (and their segmented variants) is also well understood.

The last approach is preferred in cases where the inner loop trip counts are small and/or variable, as they are for many sparse matrices. Current compilers may detect the reduction in the inner loop, but cannot recognize that a segmented reduction can be used to perform the computation in the entire loop nest. The previous chapters' recurrences parallelization technique yielded a powerful and extensible technique for reasoning about complex expression and control structure in non-nested loops. By applying the technique of loop flattening to the loop

nest, followed by applying the recurrence parallelization technique, the compiler is able to *automatically* extract this parallel solution from the serial loops.

5.3 Loop Flattening

5.3.1 Motivation

A key problem in applying the parallelization technique of the previous section is one of modelling loop nests. Ideally, we would like to be able to encode the control structure of loop nests in the functional model of recurrent loop bodies. A simple transformation, similar to loop coalescing, can be made by flattening the loop nest into a single loop in which the loop index variables are explicitly computed and checked. We achieve our goal by injecting the loop index variable computation into the recurrent loop body, so that it may be directly modelled and reasoned about using our recurrence parallelization technique.

In the case of regular loop nests, this may seem unduly complicated for the compiler. Ad-hoc changes to the analysis of the previous section might handle a reasonably large family of regular loop nests (i.e. the class of triangular iteration patterns). However, irregular loop nests are much more interesting for their powerful expressiveness. Furthermore, the technique presented here will also work well for regular loop nests.

The CSR kernel of Figure 1 is flattened below:
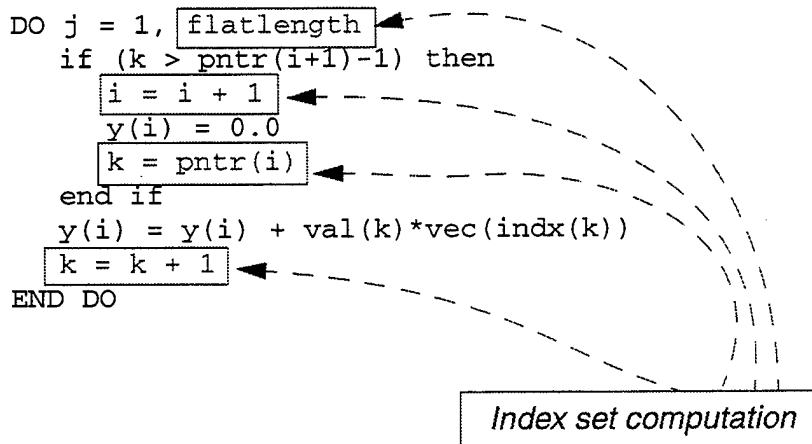
The computation for the length of the flattened loop is not shown here. Computation of the original loop index set, highlighted in the Figure, adds several instructions to increment and

73

reinitialize the loop index variables. A conditional guard is inserted around the portion of the outer loop preceding the inner loop. The condition expression checks whether the outer loop body should be executed at that point in the flattened iteration space.

The problem with this transformed loop is that a complex set of both intra- and inter-loop dependences are introduced into the loop. One approach to parallelizing this loop might try to use traditional parallelizing techniques along with our recurrence parallelizing technique to parallelize the index set computation along with the original loop body components. However, for some classes of irregular loop nests, canned preambles can be used to perform this computation. This is a better approach, since such a preamble can be hand optimized and the compiler can concentrate on amortizing the cost by hoisting it out of surrounding loops. We will discuss this further in section 5.3.5.

The segmentation of the inner loop's reduction is captured in the flattened loop's control structure, that is, the guards using the index sets computed from the original loop nest. Since the recurrence parallelization technique easily accommodates such conditional statements, we have successfully injected the original loop nest's control structure into our model for analysis.

5.3.2 The Basic Transformation

Throughout this chapter, we refer to a single loop nested within another. The techniques presented here can be extended to deeper nested loops by recursively applying the transformations. It should be noted that in most practical circumstances, irregular loop nests need only be flattened at the innermost level to expose sufficient parallelism. Thus, we refer to a simple model of nested loops, illustrated below:

```
{DO outerindex = ..., WHILE ...}
     <BODY 1>
     {DO innerindex = ..., WHILE ...}
        <BODY 2>
     END DO
     <BODY 3>
END DO
```

The underlying transformation for loop index flattening is conceptually simple. First, the total number of inner loop iterations is computed. This count is the length of the new flattened loop.

Next, the original loop indices are computed (only if they are used in the component body parts). Finally, the flattened loop is generated by placing guards around the component body parts which check whether the inner loop bounds are crossed in the original loop. This transformation is illustrated below:

```
<Compute Loop Length>       number of inner
                            loop iterations

<Compute Loop Indices>      {inner, upper}
DO flatindex = 1, flatlength
    IF (inner_loop_finished(flatindex)) THEN
        <BODY 1>
    END IF
    <BODY 2>
    IF (inner_loop_finished(flatindex+1)) THEN
        <BODY 3>
    END IF
END DO
```

The flattened loop is very similar to the original loop. Two conditional guards were introduced to check whether the outer loop body components should be invoked. However, since the inner and outer loop indices are computed prior to the execution of the flattened loop, the conditional expressions in the guards are loop invariant with respect to the flattened loop. Thus, they present no obstacles to many standard parallelization techniques. Note that although this may present an obstacle to many recurrence parallelization techniques, it presents no obstacle to the recurrence parallelization technique.

In this transformation, we must compute the flattened loop length and the original loop indices. Computing the flattened loop length in parallel can be performed using a operation over each of the inner loop lengths. For example, the following code suffices (with appropriate checks for negative or zero trip count segments and zero strides deleted):

```
flatlength = SUM((innerupper(outerlower:
$          outerupper:outerstride)
$        - innerlower(outerlower:
$          outerupper:outerstride))
$      /innerstride(outerlower:
$          outerupper:outerstride)
```

sum is simply a global sum or sum reduction. (This is actually computed as a by-product of a scan operation used when computing the inner and outer loop index sets in the next section.)

Note that the terms `innerupper`, `innerlower`, and `innerstride` in this expression may denote array expressions rather than just simple arrays. We make use of this convention frequently.

Since the original loop indices may be used by the loop body, it may be necessary to compute these. The tricky part is in parallelizing and optimizing the loop index computation; this will be discussed in more detail in the next section.

5.3.3 Counting Outer Loop Iterations

The previous discussion implicity assumes that the transformation only counts inner loop iterations in calculating the flattened loop's trip count. However, there may conceivably be instances in which the inner loop's trip count is zero while meaningful computation takes place in the outer loop's body components. It is relatively easy to adjust the basic flattening scheme and index set computation method to account for such instances. Though this method is more rigorous, in all of the cases we have examined, cases which require such an approach either never occur, or can be remedied by employing loop fission to isolate the outer loop's body components.

5.3.4 Flattening Indirection

Indirect array accesses are introduced to the flattened loop by replacing the original loop's index expressions with the precomputed indices, which are now array accesses. For some arrays, it may be profitable to copy arrays into a flattened structure for repeated references. In the case of arrays which are read, but not written, this is usually handled automatically by the backend compiler. For arrays which are written, this effect is more problematic since it obscures the true nature of the computation.

Simple array copies, reductions, and scans look like slower combining-send operations, or, worse, indeterminate and unparallelizable recurrent operations. In chapter 7, we discuss dependence recycling techniques which allow us to track such computations through such obscuring transformations. In such cases, the compiler will flatten the target array structure and insert code to copy values back into the original array structure. In the case of a sequence of flattened loops with similar inner loop structure, the lifespan of the flattened structure can be extended through all of the flattened loops to eliminate the costs of copying.

An example of this effect:

```
do i = 1, N
   do k = p(i), p(i+1)-1
      y(i) = y(i) + b(k)
   end do
end do
```

The flattened loop with indirection looks like superficially like a combining-send operation:

```
do f = 1, flatlen
   y(i(f)) = y(i(f)) + b(k(f))
end do
```

Since the original operation was a reduction, the compiler can infer that this flattened operation is actually a segmented reduction. To realize this operation in the transformed code, we replace the original array by the flattened array, and then copy the appropriate array elements back to original array (`lengths` is the prefix sum of the inner loop lengths):

```
do ip = 1, N                                      Initialize
   y_f(lengths(ip)+1) = y(ip)                     flattened
end do                                            structure

do f = 1, flatlen
   if (innerlooptripped) then                     Flattened
      y_f(f+1) = y_f(f) + b(k(f))                 loop
   endif
end do

do ip = 1, N                                      Copy values
   y(ip) = y_f(lengths(ip+1)) + b(k(lengths(ip+1)))   back into
end do                                            original array
```

Once again, details of how to deal with zero length inner loop trip counts are elided for clarity. Note that in the case of uninitialized structures the initialization of the flattened array may be either eliminated.

An important consideration in employing this scheme is the amount of space used by the flattened data structure. Fortunately, the space used here will be required for use by the recurrent primitive templates which implement the computations in these loops. For example, in detecting a segmented reduction or scan, temporary space proportional to the flattened loop's trip count is needed.The amount of space required is precisely the amount of space effectively allocated by this array index flattening scheme.

5.3.5 Optimizing and Amortizing Index Set Computation

The simplest approach to computing the original loop indices is to use the original loop nest as a template for computing the loop index sets serially. This computation may be amortized over an enclosing loop (as in conjugate gradients), allowing us to reap the benefit of the transformation to a flattened loop. The next best approach is to have the compiler try to parallelize the index set computation. These options are used only as stopgap measures, since we have developed methods to parallelize the index computation portion of one of the more important and common classes of irregular loop nests.

The type of loop nest the implementation currently handles is referred to as *segmented*. In segmented loop nests, the inner loop bounds are indirect accesses of arrays using the outer loop indices, but are loop invariant, as in the case of the CSR sparse matrix-vector multiplication. For this class of irregular loops, there is a general parallel template which may be used to parallelize the loop. For readability, we have left out checks for zero or negative segment lengths and zero step sizes:

```
C  Compute segment lengths
   LEN_VEC = +_SCAN((
  $  INNERUPPER(OUTERLOWER:OUTERUPPER:OUTERSTRIDE)-
  $  INNERLOWER(OUTERLOWER:OUTERUPPER:OUTERSTRIDE))/
  $  INNERSTRIDE(OUTERLOWER:OUTERUPPER:OUTERSTRIDE))

C  Compute outer indices at each point in the

C  flattened loop

   OUTER_INDEX(1:LEN_VEC(OUTERUPPER)) = 0
   OUTER_INDEX(LEN_VEC) = OUTERSTRIDE
   OUTER_INDEX(1) = OUTERLOWER
   OUTER_INDEX = +_SCAN(OUTER_INDEX)

C  Compute inner indices at each point in the

C  flattened loop

   FLAG_VEC(1:LEN_VEC(OUTERUPPER)) = 0
   FLAG_VEC(LEN_VEC) = 1
   INNER_INDEX = INNERSTRIDE(OUTER_INDEX)
   INNER_INDEX(FLAG_VEC) =
  $  INNERLOWER(OUTERLOWER:OUTERUPPER:OUTERSTRIDE)
   INNER_INDEX = +_SEGSCAN(INNER_INDEX, FLAG_VEC)
```

The first step is to compute the loop trip counts for the inner loop (essentially, the segment lengths). Using these, computing the inner and outer loop indices is easy: the outer loop index
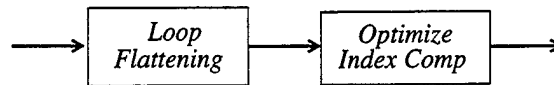
Figure 5.2 Loop flattening pass.

is incremented at the beginning of every segment; the inner loop index is initialized to the proper lower bound at the beginning of every segment, and incremented otherwise.

To improve the performance of the flattened loop, we perform certain optimizations on this code and the flattened loop:

- *Hoist the index set computation out of enclosing loops.* This effectively amortizes the overhead of the flattened loop over multiple invocations. This is analogous to amortizing the setup time for a specialized sparse matrix-vector multiplication library function. Standard data-flow analysis techniques can be employed to achieve this.

- *Compute only the index sets necessary.* For example, if either the outer or inner loop indices are not necessary, we can save execution time and/or memory usage.

- *Perform run-time tests to eliminate continuous, constant stride index sets with flattened loop index expressions.* This is essentially induction variable detection, but with a twist. Certain properties of the inner loop bounds can be checked symbolically at compile time. One property is to check whether the upper and lower bounds are symbolically continuous (i.e. Is the expression **INNERLOWER**(*outerindex* + *1*) equivalent **INNERUPPER**(*outerindex*)?) Another is to check that the inner stride is constant. Given this, if we can verify at run-time that the bounds for the inner loop (**INNERLOWER** and **INNERUPPER**) are monotonic, then we can replace occurrences of the inner index by simple affine expressions of the new flattened index expression. In this way, potentially costly indirection can be eliminated as well as potentially more expensive primitive selection in later compiler phases. The current approach to this is to generate two versions of the flattened loop, one optimized and one unoptimized in this manner, to be selected by the run-time test.

5.3.6 Compiler Pass Architecture

The pass for this transformation is structured simply as loop flattening, followed by optimizations of the index computation, in figure 5.2. The loop flattening pass does not flatten past a nesting level of two, though it can recursively flatten deeper loop nests. In our practical experience, interesting irregular loop nests of depth greater than two have not arisen. The compiler

then attempts to eliminate indirection and hoist index computation out of surrounding loop levels. Note that no real parallelization takes place in loop flattening. Rather, it is a preprocessing step to enable later phases of parallelization.

5.4 Example Code: Sparse Matrix Vector Multiplication

The CSR and CSC kernels were compiled using our parallelizing techniques, and compared against the best code generated by the CF77 compiler. In compiling for a single head, the CF77 compiler vectorizes the reduction in the inner loop of CSR. In compiling for multiple heads, the CF77 compiler tasks the outer loop of the kernel and vectorizes the inner loop reduction of CSR. For this kernel, our compiler generates a segmented reduction, whose pseudo-Fortran is below, which is simultaneously tasked and vectorized:

```
flat = pntr(N+1) - 1
   vecwork(1:flat) = val(1:flat:1) *
   $                      vec(indx(1:flat))

condition = k(1:flat).le. pntr(i(1:flat)+1) - 1

y(1:N) = APPLY(FUN_REDUCE(
   $              {λx → (condition?(x + vecwork):vecwork)},
   $              flatlength,
   $              (i(1:flat - 1) .ne. i(2:pntr(N+1)-1))),
   $           Y(1:N))
```

The brackets on the lambda expressions denote the analyzed version of the enclosed loop modeling function. The third argument to the function composition reduction (FUN_REDUCE) is simply shorthand for a pack of the results. The CF77 compiler only vectorizes the inner loop of the CSC kernel on both single and multiple head configurations. Our compiler generates a combining-send operations which is simultaneously tasked and vectorized, below:

```
vecwork(1:flat) = val(1:flat:1) *
   $                      vec(i(1:flat))

y(1:N) = APPLY(FUN_COMB_SEND(
   $              {λx → x + vecwork},
   $              i(1:flat)
   $              flat,
   $              N),
   $           Y(1:N))
```

5.5 Review

A framework for compiling irregular loops was presented. It enables parallelization by transforming nested iterative structure to nested conditional structure, which the recurrence parallelization technique can handle well.   The technique will further prove to be useful in the next chapter, when we start to general irregular loop nests from divide-and-conquer algorithms.

Chapter 6

Irregular Control Structure II - Control Embedding

In this chapter we discuss a technique to enable the parallelization of divide and conquer style recursive functions. In addition to the fundamental control structure transformation, we introduce new techniques for gathering accurate dependence information in the presence of partitioning code and recursion.

6.1 Divide and Conquer Recursion

Recursion is an important mechanism for the expression of algorithms written in a divide-and-conquer style. Algorithms in sorting [4], computational geometry [68][10], and mesh generation often use divide-and-conquer. In serial imperative languages, recursive subroutine calls are usually invoked upon partitioned problems. This is an elegant and concise control mechanism for repeatedly invoking the same basic algorithm on ever smaller partitions of the problem. Using iterative structure is not feasible because of the syntactic complexity that such code might introduce. From a parallelization perspective, this significantly complicates the task of both manually and automatically parallelizing such code.

Examine the generic divide-and-conquer recursive subroutine and its *dynamic call graph* (DCG) in figure 6.1. It is probably relatively easy for both the programmer and the compiler to parallelize the body components of the recursive subroutine. Furthermore, with a bit more effort the user can parallelize the recursive calls, though it will be difficult for the compiler without some of the analyses presented in this chapter. Unfortunately, as can be seen in the hypothetical DCG for this code, the amount of parallelism available in the loop body will decrease as the partitions grow
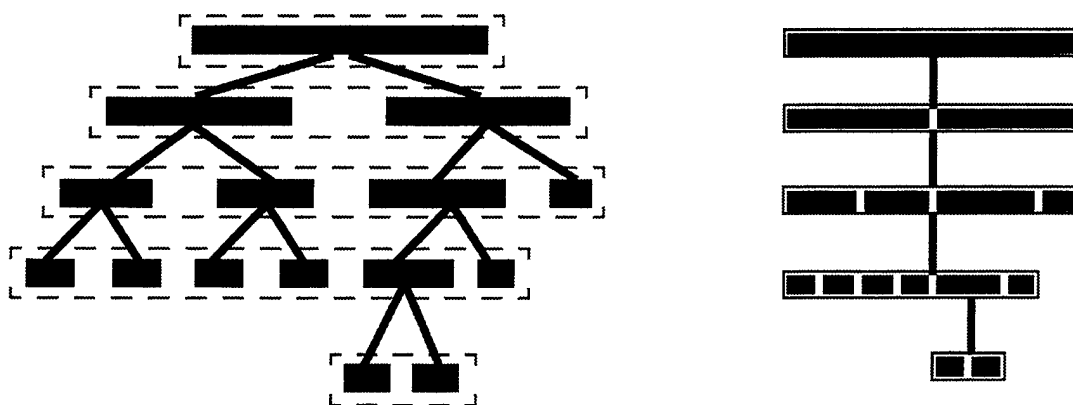
```
subroutine recurse(...)          subroutine recurse_embedded(...)
PreBody                          for each partition
for each partition                  PreBody
   call recurse(...)             endloop
endloop                          call recurse_embedded(...)
PostBody                         for each partition
                                    PostBody
                                 endloop
```

Figure 6.1 A prototypical divide-and-conquer algorithm and its control embedded version.

ever smaller. Furthermore, the partition sizes may vary, making load balancing a much more difficult task for the user. Such irregularity is inherent in divide-and-conquer algorithms with data dependent partitioning strategies.

There is parallelism within the subroutine and across subroutine call boundaries here. We can exploit both simultaneously in an automatic parallelizing compiler by making the following observations. The recursive calls are surrounded by an implicit or explicit loop. This loop can be *embedded* within the recursive call so that rather than recursive over each partition, a single recursive call performs the subroutine body for each partition. The resulting code and DCG is in figure 6.1. The advantage here is that by compiling and exclusively parallelizing the body component of the embedded version of the subroutine, we can effectively automatically parallelize across all sibling recursive calls. This means that the compiler can exploit all available parallelism at each level of the DCG.

```
        │
┌──────────────────────┐
│ 6.2 - Modified Depen- │
│                      │
│   dence Analysis     │
└──────────────────────┘
        │
        │
        ▼
┌──────────────────────┐       ┌──────────────────────────────────┐
│ 6.4 - Preprocessing   │ ─ ─ ─ │  Eliminating Returns - 6.4.1      │
│                      │       │                                  │
│       Steps          │ ─ ─ ─ │  Winnowing Recursive Calls - 6.4.2│
└──────────────────────┘       └──────────────────────────────────┘
        │
        │                      ┌──────────────────────────────────┐
        │                ─ ─ ─ │  Embedding Loops - 6.5.1          │
        ▼                      │  Privatizing Local Variables - 6.5.1.1│
┌──────────────────────┐       │  Expanding Parameters - 6.5.1.2   │
│  6.5 - Control        │       │  Embedding Conditionals - 6.5.2   │
│                      │ ─ ─ ─ └──────────────────────────────────┘
│   Embedding          │
└──────────────────────┘
        │
        ▼
```
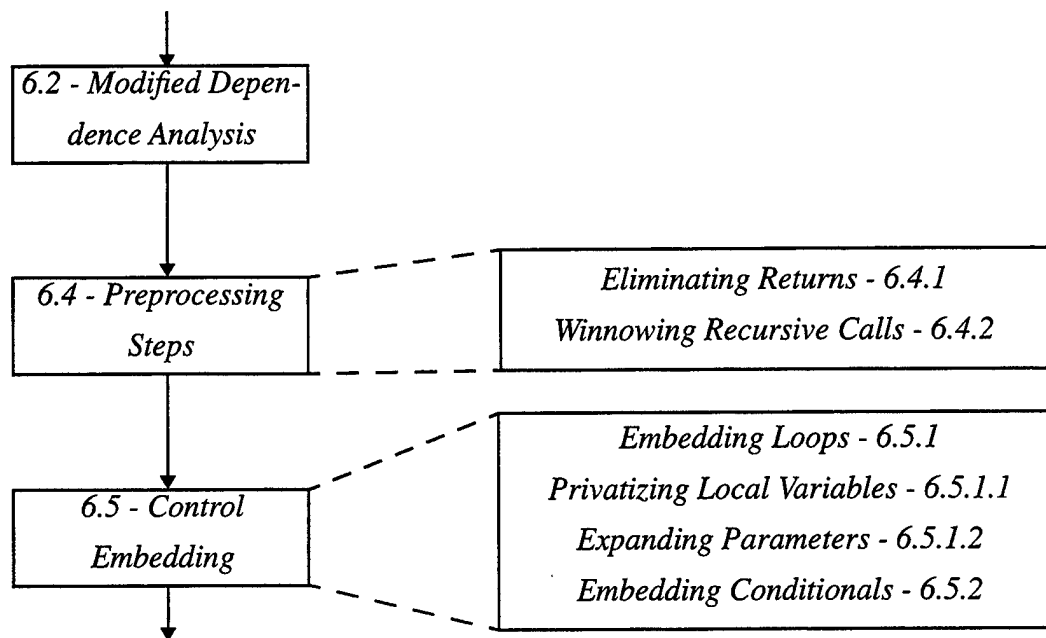
Figure 6.2 Components required for control embedding in recursive subroutine calls with corresponding section numbers.

6.2 Control Embedding

The fundamental mechanism for converting interprocedural to intraprocedural parallelism is through control embedding. The control we embed is precisely that control which manages the interprocedural parallelism in these algorithms. In the case of divide and conquer algorithms, the control structure we are interested in manages iterations over partitions, as well as conditions for completion of the algorithm. This means that we would like to embed loops around recursive calls, either explicitly or implicit. It also means that we would like to embed conditional structure surrounding or otherwise affecting (i.e. through a return statement) the recursive calls.

Conceptually, embedding control is simple. Loops and conditionals are folded into function calls which they surround, as in figure 6.1. The difficulty is in creating the circumstances in which the transformation can be applied and in managing the parameter space. In particular, we would like to isolate the recursive calls and their surrounding control structure. Furthermore, we need to employ semantics-preserving transformations to eliminate difficult constructs such as `return` statements, as well as marshalling actual parameters and privatizing local variables for each sibling recursive call.

This chapter begins with a description of a modification of a typical symbolic analysis to support the generation of dependence information necessary for the transformation to be legally applied. Existing analyses lack the ability to find precise relationships between induction variables whose values are alternately modified. In partitioning loops of divide-and-conquer algorithms, this relationship is important for determining that work between partitions is independent.

Subsequent sections describe preprocessing steps in the transformation to simplify control embedding. The two transformations here, return elimination and recursion winnowing, have as a goal the transformation of any divide-and-conquer style recursion to the prototypical recursive subroutine that had control embedded in figure 6.1. This step facilitates the application of control embedding by effectively isolating that control which must be embedded in the recursive subroutine.

Control embedding for recursive subroutines is the last transformation step discussed in this chapter. The basic idea behind facilitating embedding loops for recursive loops is to find a fixed point for the embedded loops. The transformation also includes mechanisms for embedding control structure surrounding the recursive calls, but nested within the embedded loops. Thus, we refer to this transformation as the more general 'control' embedding rather than 'loop' embedding. The loop embedding process also necessitates the promotion of parameters to simultaneously track the multiple activation records which are, in effect, being simultaneously emulated. Finally, local parameters may be expanded in a privatization effort to facilitate parallelization.

6.3 Dependence Analysis and Monotonic Induction Variables

In the context of dependence analysis, regular induction variables are easy to handle. The compiler simply substitutes the equivalent expression of the loop index variable and performs traditional dependence analysis. However, there are several other varieties of induction variables [94], some of which are used to partition data in divide-and-conquer algorithms.

The type which are of greatest interest to us are *monotonic induction variables*. These are induction variables which are incremented by values of the same sign, but not always by the same value. The variable j in the following packing code is such a variable because the conditional statement implies that the value may not always be incremented:

```
j = 1
do i = 1, n
   if (f(a(i))) then
      b(j) = a(i)
      j = j + 1
   end
enddo
```

The fact which is of importance here is that the monotonic variable guarantees that the data movement performed here is a *permute*, i.e. there is no write conflict. In this instance, a pack function or intrinsic can be used to parallelize this loop. However, we will point out that this simple observation is not sufficient for more complicated codes. In particular, we are interested in the kind of partitioning code that frequently occurs in divide-and-conquer algorithms. Here is an example of such a code:

```
lower = begin
upper = end
do i = begin, end
   if (f(a(i))) then
      b(lower) = a(i)
      lower = lower + 1
   else
      b(upper) = a(i)
      upper = upper - 1
   endif
enddo
```

Either lower or upper individually can be used in *packing* or *gather* operations, but to maintain the serial semantics of this code, one may not employ such a strategy. The problem is that the partitions defined here might overlap with each other. However, in this particular loop, we know that this is not the case. We would like the compiler to understand this as well.

We present extensions to a symbolic analysis scheme [87] which uses an extension of Single Static Assignment (SSA) form [30] called Gated Single Assignment (GSA) form [11]. SSA form is useful for induction variable recognition [94] and many symbolic analyses [87].

The essential idea is to try to encapsulate the guarding of these monotonic induction variables with conditionals in the affine constraints. For induction variables, their values are usually computable or constrained sufficiently in closed form equations of loop index variables. Unfortunately, no accurate closed form representation in the loop indices can be found which

in existing analyses sufficiently reflects the interrelationship between the monotonic variables. We have developed a mechanism for injecting the proper terms and constraints into the symbolic analysis that works quite naturally on Gated Single Assignment form.

The topic discussed in this subsection essentially provides information necessary for the transformation introduced in this chapter to be legally and confidently applied by the compiler. The background work discussed here is essentially derived or directly based on the representations and analysis schemes developed in Peng Tu's Ph.D. dissertation [87]. Any deviations from that work will be noted.

6.3.1 Gated Single Assignment Form

In SSA form for straightline code, exactly one definition of a variable reaches each use of that variable. Each variable defined is given a new, unique name each time it is defined. To resolve multiple definitions from differing paths joining in the control flow graph in more complex code, φ-*functions* are used to resolve which value is used. We will use the following code example to illustrates the SSA and GSA forms:

```
lower = begin
upper = end
do i = begin, end
    if (P) then
        lower = lower + 1
    else
        upper = upper - 1
    endif
enddo
```

The SSA form for this code segment is:

```
lower₁ = begin
upper₁ = end
do i = begin, end
    lower₂ = φ(lower₁, lower₄)
    upper₂ = φ(upper₁, upper₄)
    if (P) then
        lower₃ = lower₂ + 1
    else
        upper₃ = upper₂ - 1
    endif
    lower₄ = φ(lower₃, lower₂)
```

```
    upper₄ = φ(upper₃, upper₂)
enddo
lower₅ = φ(lower₁, lower₄)
upper₅ = φ(upper₁, upper₄)
```

The insertion of φ-functions here indicate where multiple possible values for a particular variable converge. This is most often the case in join nodes of the control flow graph. In this example, there were φ-functions inserted at the head of loops, immediately following a loop, and after a conditional branch in which the variable is modified. This guarantees that only a single definition reaches each use of a variable.

The GSA form preserves the essential reaching definition properties of SSA, but replaces the φ-function in some cases with several new gating functions:

- γ functions replace φ functions at join of control paths from differing conditional branches. They also include the predicate for the conditional statement.

- μ functions replace φ functions at the head of a loop. It includes the exit conditions or index variable ranges for the loop.

- η functions replace φ functions at the exit of a loop.

From a program analysis point of view, these preserve the looping and conditional structure of the program while retaining the useful SSA-like properties. The importance of these GSA features becomes evident in performing symbolic analysis to determine value ranges computed in loops and conditional branches.

The running example, rewritten in GSA form:

```
lower₁ = begin
upper₁ = end
do i = begin, end
    lower₂ = μ((i=begin,end),lower₁,lower₄)
    upper₂ = μ((i=begin,end)upper₁,upper₄)
    if (P) then
        lower₃ = lower₂ + 1
    else
        upper₃ = upper₂ - 1
    endif
    lower₄ = γ(P, lower₃, lower₂)
    upper₄ = γ(P, upper₂, upper₃)
enddo
```

```
lower₅ = γ(end < begin, lower₁, η((i > end), lower₄))
upper₅ = γ(end < begin, upper₁, η((i > end), upper₁, upper₄))
```

The use of the η functions in the last two statements deserves some explanation. If the source language guaranteed non-zero trip counts, the use of γ functions would not be necessary in uses of η functions. In this case, the γ functions cases handle the possibility in Fortran of zero trip counts.

Fast and efficient algorithms for constructing both SSA and GSA forms are presented elsewhere [30][87], and their description lies outside the scope of this thesis.

6.3.2 Symbolic Analysis

Symbolic analyses typically entail manipulating and propagating symbolic expressions representing values computed in the program being analyzed. This is useful for, among other things, proving assertions in the program and finding symbolic value range information for performing more accurate dependence analysis. For example, the predicate in an `if` statement can be used as a symbolic assertion by the compiler in analyzing its branches.

There are two mechanisms by which symbolic expressions are propagated. *Symbolic forward substitution* propagates symbolic expressions for variables forward in the program, in a manner akin to the symbolic execution of the program. The process is similar to constant propagation. The problem with this approach is that the size and number of expressions being propagated becomes very large, while the portion of those expressions which are relevant to a particular compiler goal may be small. Furthermore, it is typically not necessary to express all symbolic values in terms of program inputs.

Symbolic backward substitution starts at an expression and symbolically substitutes symbolic expressions for previous definitions of its arguments. The advantages of this approach is that it can be more easily tailored to satisfying a particular goal. The starting point for the analysis is determined by the compiler objective and the backwards substitution process can be stopped when the objective is achieved. This kind of *demand-driven symbolic analysis* is very effective for the kind of dependence analysis we are interested in here. Again, the discussion here is based heavily on Tu's dissertation [87] unless otherwise noted.

We use the following code segment to illustrate backward substitution:

```
upper₁ = end
if (P) then
   ...
else
   upper₃ = upper₂ - 1
endif
upper₄ = γ(P, upper₂, upper₃)
```

Here, we are interested in the upper bound for the variable $upper_4$. Substituting backwards gives us:

```
upper₄ =   γ(P, upper₂, upper₃)
           γ(P, upper₂, upper₂ - 1)
           γ(P, end, end - 1)
```

This gives us an upper bound of ($upper_4 \leq end$).

The symbolic expression *(SE)* for the value of a variable may be composed of multiple instances of the three gating functions here. It is useful to be able to compute a qualified SE for the variable(s) of interest in different control context. To that end, we describe the notion of path projection to determine the path-restricted values *(PV)* for the *SE* given the control flow conditions *(PC)* leading to that particular control path. We compute the projection *PV = SE(PC)* as follows:

$$SE(PC) = SE \qquad \qquad \textit{if SE contains no gating functions}$$

$$\gamma(P, V_t, V_f)(PC) = V_t(PC) \qquad \textit{if } PC \supset P$$

$$\gamma(P, V_t, V_f)(PC) = V_f(PC) \qquad \textit{if } PC \supset \neg P$$

$$\gamma(P, V_t, V_f) = \gamma(P, V_t(PC), V_f(PC)) \qquad \textit{otherwise}$$

$$\mu(L, V_{init}, V_{iter})(PC) = \mu(L, V_{init}(PC), V_{iter}(PC))$$

$$\eta(P, V)(PC) = V(P \wedge PC)$$

As an example, we consider the previous code segment. Recall that the resulting symbolic value we computed using backward substitution was $upper_4 = \gamma(P, upper_2, upper_3)$. Assume that we reach a point later in the code with $PC = \neg P$, such as in the false branch of this conditional block:

```
if (P) then
    ...
else
    ...= upper₄
endif
```

Using path projection, we can compute that the value of $upper_4$ at this point is:

$$upper_4 = \gamma(P, upper_2, upper_3)(\neg P)$$
$$= upper_3(\neg P)$$
$$= upper_3$$

6.3.3 Extensions for Interrelated Monotonic Induction Variables

Unfortunately, this symbolic analysis has some shortcomings in dealing with the monotonic variables seen in the first example of section 6.3.1. Applying the symbolic analysis introduced in the last section to determine the values of $lower_2$ and $upper_2$:

```
lower₂ = μ((i=begin,end),begin, γ(P, lower₂ + 1, lower₂))
upper₂ = μ((i=begin,end), end, γ(P, upper₂, upper₂ - 1))
```

To determine the dependence of two accesses that use these two variables, the compile must computer whether $upper_2 \cap lower_2 = \varnothing$. One conservative estimate is to simply derive the maximum and minimum values of the variables. Along with the monotonicity property, we can prove that the two do not overlap if the ranges are disjoint. We can determine the maximum and minimum values of these variables using these symbolic expressions:

```
max(lower₂)    ≤ max(μ((i=begin,end),begin, γ(P, lower₂ + 1, lower₂)))
               = μ((i=begin,end),begin,max(γ(P, lower₂ + 1, lower₂)))
               = μ((i=begin,end),begin, lower₂ + 1)
min(lower₂)    ≥ min(μ((i=begin,end),begin, γ(P, lower₂ + 1, lower₂)))
               = μ((i=begin,end),begin,min(γ(P, lower₂ + 1, lower₂)))
               = μ((i=begin,end),begin, lower₂)
max(upper₂)    ≤ max(μ((i=begin,end), end, γ(P, upper₂, upper₂ - 1)))
               = μ((i=begin,end), end, max(γ(P, upper₂, upper₂ - 1)))
               = μ((i=begin,end), end, upper₂)
min(upper₂)    ≥ min(μ((i=begin,end), end, γ(P, upper₂, upper₂ - 1)))
               = μ((i=begin,end), end, min(γ(P, upper₂, upper₂ - 1)))
               = μ((i=begin,end), end, upper₂ - 1)
```

From these expressions, we can directly infer the ranges of values for variables:

```
begin ≤ lower₂ ≤ end
```

```
begin ≤ upper₂ ≤ end
```

The problem here is that the value ranges for `lower₂` and `upper₂` overlap. While the ranges are useful for determining whether the differing invocations of the containing block overlap, they are insufficient for determining whether there is output dependence in the loop body between the two branches. Unfortunately, the symbolic boundaries in these ranges cannot be tightened further. However, the fact that the conditional expression in the if-branch constrains the iterations that the two variables are incremented or decremented to be disjoint can be exploited to prove, among other things, that they do not overlap and there is no output dependence.

If the compiler recognizes that the if-statement's branches essentially partition the loop's trip count so that neither variable is incremented or decremented in the same iteration, then the dependence analyzer can build a constraint reflecting the interdependence between the values of the variables in question.

We construct a filter to coalesce affine constraints for symbolic values as well as tag expressions with trip count information. The operator we define to construct these constraints is called *TC*. This operator adds explicit variables to reflect trip counts and their partitioning by conditional branches. It also generates the relevant constraints.

When applied to μ nodes, the *TC* operator multiplies the current running trip count with the trip count of the loop it is applied to. It also generates the necessary constraints for computing the symbolic value of the trip count, as well as the constraints on the loop index:

$$TC(T, \mu((index=lower,upper,stride),init,loop)) =$$
$$\mu((index=lower,upper,stride),TC(T \times T,init), \ TC(T \times T,loop))$$
Generates constraints:
$$(lower \leq index \leq upper, \ \exists s, stride \times s = index ,$$
$$T = \frac{upper - lower}{stride} + 1)$$

When applied to γ nodes, the *TC* operator splits the current running trip count for each branch by adding a new variable whose size is constrained to be less than or equal to the total trip count (but greater than 0). The idea here is to symbolically represent the time a conditional spends in each of its branches. Thus, the new variable represents the portion of the enclosing loops' trip cumulative trip counts spend in each of the conditional's branches:

```
TC(T, γ(P, tval, fval)) =
      γ(P, TC(C_PT,tval), TC(T - C_PT,fval))
   Generates constraints:
      (0 ≤ C_PT ≤ T)
```

The variables T and $C_{P,T}$ are newly generated only if the index range and index range-predicate pair have not been previously encountered. Not that this implies that for γ gates that predicates will have to be tested for equality, though, in the cases we are interested, variations on simple syntactic equality tests should suffice.

This provides the necessary constraint-based relationships between monotonic variables. Operators such as max and min can now be applied to expressions with nested *TC* operators in a straightforward manner. Consider the application of *TC* to lower$_2$ and upper$_2$:

```
TC(lower₂)  = TC(μ((i=begin,end),begin, γ(P, lower₂ + 1, lower₂)))
            = μ((i=begin,end),TC(1,begin),
                      TC(end-begin, γ(P, lower₂ + 1, lower₂)))
        Generates constraints:
            (0 ≤ C ≤ end-begin)
            = μ((i=begin,end),TC(1,begin),
                      γ(P, TC(C,lower₂ + 1), TC(end-begin-C,lower₂)))
TC(upper₂)  = TC(μ((i=begin,end), end, γ(P, upper₂, upper₂ - 1)))
            = μ((i=begin,end),TC(1,end),
                      TC(end-begin+1, γ(P, upper₂, upper₂ - 1)))
            = μ((i=begin,end),TC(1,end),
                      γ(P, TC(C,upper₂), TC(end-begin+1-C,upper₂ - 1)))
```

Note that we have represented the loop trip count generated for the μ gate directly rather than using a separate variable, for clarity. We compute the ranges for these variables:

```
min(lower₂) = min(μ((i=begin,end),TC(1,begin),
                      γ(P,TC(C,lower₂ + 1),TC(end-begin+1-C,lower₂))))
            = μ((i=begin,end), begin,
                  min(γ(P,TC(C,lower₂ + 1),TC(end-begin+1-C,lower₂))))
            = μ((i=begin,end),begin,TC(end-begin+1-C,lower₂))
            = μ((i=begin,end),begin,lower₂)
            = begin
max(lower₂) = max(μ((i=begin,end),TC(1,begin),
                      γ(P, TC(C,lower₂+1), TC(end-begin+1-C,lower₂))))
            = μ((i=begin,end),begin,
                  max(γ(P, TC(C,lower₂+1),TC(end-begin+1-C,lower₂))))
            = μ((i=begin,end),begin,TC(C,lower₂ + 1))
            = begin + C
min(upper₂) = min(μ((i=begin,end),TC(1,end),
                      γ(P, TC(C,upper₂), TC(end-begin+1-C,upper₂-1))))
```

94

$$= \mu((\text{i=begin,end}),\text{end},$$
$$\min(\gamma(P,TC(\text{C, upper}_2),\ TC(\text{end-begin+1-C,upper}_2-1))))$$
$$= \mu((\text{i=begin,end}),\ \text{end},\ TC(\text{end-begin+1-C,upper}_2-1))$$
$$= \text{begin} + 1 + \text{C}$$
$$\max(\text{upper}_2) = \max(\mu((\text{i=begin,end}),TC(1,\text{end}),$$
$$\gamma(P,\ TC(\text{C,upper}_2),\ TC(\text{end-begin+1-C,upper}_2-1))))$$
$$= \mu((\text{i=begin,end}),\text{end},$$
$$\max(\gamma(P,\ TC(\text{C,upper}_2),\ TC(\text{end-begin+1-C,upper}_2-1))))$$
$$= \mu((\text{i=begin,end}),\text{end},TC(\text{C,upper}_2))$$
$$= \text{end}$$

So, the resulting constraints are:

```
begin ≤ lower₂ ≤begin + C
begin + C + 1 ≤ upper₂ ≤ end
0 ≤ C ≤ end-begin (generated by TC)
```

It is easy to see these constraints prove that `lower`$_2$ and `upper`$_2$ do not overlap.

6.4 Preprocessing Steps

The first step in embedding control in recursive subroutines is to isolate those control structures we will embed. Recursive subroutine calls may be embedded, along with other code, in loops and conditional constructs. Isolating the calls with their surrounding control structure provides a concise description of the number of recursive calls or partitions and conditions for liveness of the partitions. We also transform return statements into conditional statements, so that embedding loops can correctly be applied. This also serves to create embedable conditions in which partition workload is implicitly tracked.

6.4.1 Eliminating Returns

The presence of return statements in a subroutine presents several challenges to use. Return statements are difficult to model in the loop modeling functions employed by the underlying recurrent loop analysis. Return statements can also rule out the applicability of control embedding process. Embedding a loop around a return statement in the subroutine body affects subsequent loop iterations, consequently affecting all subsequent partitions be worked on. In contrast, the return in the non-embedded loop body would only affect the particular problem partition being worked on.

Returns can be eliminated by employing conditional constructs. The essential idea is to surround code that is affected by a return statement by placing it within if-statements with condi-

tions which lead to the execution of the return. Several observations motivate the use of conditionals in this case:

- Conditional constructs are easily represented in loop modeling functions for recurrent loop analysis.

- Conditional constructs can be embedded within recursive calls

The analysis employed here propagates the conditions under which control flow reaches return statements, The code we are interested does not have returns embedded within any kind of looping structure. Furthermore, we are not interested in solving the problem over arbitrary control flow graphs, though a data flow formulation for more general control flow graphs is possible. Thus, the analysis here focuses on structured flow graphs with loops collapsed into single nodes.

In this analysis, there are two variables used to compute path conditions for returns. PC_{local} tracks any contribution to the path conditions by a basic block. PC contains the accumulated path condition resulting in prior execution of a return statement.

$PC_{local}(B) = C$ *if B is a conditional node in the CFG and C is the condition*

$PC_{local}(B) = TRUE$ *otherwise*

$$PC(B) = \bigcup_{p \in pred(B)} (PC(p) \wedge PC_{local}(p)),$$

    where $\bigcup = \vee$ *(logical-or).*

We can now uses these path conditions to propagate return conditions. RC_{local} contains the local return condition if any return is present in the block. RC contains the set of return conditions propagated to the block and is what we will use to guard execution of the block.

$RC_{local}(B) = PC(B)$ *if B contains a Return statement*

$RC_{local}(B) = \varnothing$ *otherwise*

$$RC(B) = \bigcup_{p \in pred(B)} (RC_{local}(p) \cup RC(p)),$$

    where $\bigcup = \cup$ *(set union).*

We remove return statements by first removing all statements that follow return statements in the same basic block. Then we delete the return statement. The final step is to insert the return

guards for the blocks. The conditions we use in the if-blocks is simply $\neg RC(B)$. However, note that simply inserting a guard for every subsequent block is likely to generate many redundant and nested if-blocks. It suffices to simply insert one guard in such cases. To eliminate this redundancy, we employ another simple analysis to determine whether to insert a guard or not. *RGB* notes whether a return guard needs to be inserted prior to the block.

$$RGB(B) = \bigcup_{p \,\in\, pred(B)} (RC(B) \neq RC(p)),$$

    where $\bigcup = \vee$ *(logical-or).*

We would also like to eliminate redundancy in conditional expressions in the return guards. To that end, we keep track of the current running conditional expression effectively guarding the current nesting level(s). RCL_i tracks the return conditions for which guards have been inserted at nesting level i. The code for computing this variable and generating the return guards is listed below:

$RCL_0 = \varnothing$
for each $B \in CFG$ in depth-first order
    if (B is a join node) then
        insert $IFCOUNT_{LEVEL}$ endif-stmts
        LEVEL = LEVEL - 1
    endif
    if (B is a split node) then
        LEVEL = LEVEL + 1
        $RCL_{LEVEL} = RCL_{LEVEL-1}$
        $IFCOUNT_{LEVEL} = 0$
    endif
    $RC_{refined}(B) = RC(B) - RCL_{LEVEL}$
    $RCL_{level} = RC(B)$
    if RGB(B) then
        insert if-stmt prior to block with $RC_{refined}(B)$
        $IFCOUNT_{LEVEL} = IFCOUNT_{LEVEL} + 1$
    endif
end for

6.4.2 Threshing Recursive Calls

Isolating recursive subroutine calls is important in identifying and embedding surrounding control structure for the recursive calls. The process by which we achieve this is by repeatedly applying a generalized loop fission transformation distributes both conditional and iterative control structure around its statements. In a process we liken to *threshing*, we apply this transformation repeatedly to blocks of code that contain recursive code, proceeding from innermost control structure outward, in an effort to isolate the recursive calls.

The fission transformation takes an arbitrary piece of code and an arbitrary specification of the desired clustering of the statements. By cluster, we mean which statements will remain together when control structure is distributed. Only outer level control is distributed, so that if finer grain distribution is desired then the routine must be applied recursive from inner loops or conditional blocks outward. The routine performs any necessary scalar expansion resulting for the chosen clustering. We will refer to the routine as *fission_by_cluster* in the following algorithm descriptions.

The basic outline of the threshing algorithm is as follows. (Again, we are only interested in structured code.) It first identifies which control blocks contain recursive calls. Other control blocks, such as do-blocks or if-blocks, are abstracted out as single statements since they will invariably remain in the same relative position. It then recursively finds the innermost control blocks containing recursive calls. It identifies the recursive calls and designates them as the middle cluster. The first and last clusters are those statements which must precede or follow the recursive calls, as determined by the data dependence conditions. The resulting clusters of statements are abstracted as three single statements for the remainder of the analysis. The algorithm continues by returning to the surrounding outer level of nesting and performing the clustering-fission scheme again, and so forth.

A high-level pseudo-code description of algorithm is given below. *S* is a set of individual statements and control blocks abstracted into statements. A subroutine call on a control block operates on the statements and control blocks enclosed immediately within. It is implicitly assumed that *fission_by_cluster* abstracts the clusters into statements:

thresh_block(Statements S)

    if (not innermost(S)) then

        foreach (s ∈ S)

if (control_block(s) and contains_recursive_call(s)) then

    thresh_block(s)

  endif

endif

$cluster_2 := \{s : s \in S \ and \ contains\_recursive\_call(s)\}$

$cluster_1 := \{s : s \in (S - cluster_2) \ and \ precede\_by\_dep\_info(s, \ cluster_2)\}$

$cluster_3 := \{s : s \in (S - cluster_2) \ and \ follow\_by\_dep\_info(s, \ cluster_2)\}$

if $(cluster_1 \cap cluster_3) \neq \varnothing$ *then abort_tranformation() endif*

$S := fission\_by\_cluster(S, \ cluster_1, \ cluster_2, \ cluster_3)$

end

6.5 Embedding Control

Upon isolating control structure around recursive calls and eliminating returns, we have something similar in control structure to the prototypical divide-and-conquer subroutine in figure 6.1. We can now embed that isolated control structure in the recursive calls. On the surface, this seems a fairly straightforward proposition until the issue of recursion is thrown into the mix. We will examine the issues and techniques of embedding iterative control and conditional control in turn.

6.5.1 Embedding Loops

The loop embedding transformation simply takes surrounding iterative structure and embeds it within a subroutine call [39]. The essential idea is to expose the subroutine body to the parallelism created by any surrounding loops. Here is a simple example:

```
do i = 1, n
   call mult(a(i),b(i))
enddo

subroutine mult(a,b)
real a, b
a = a*b
end
```

Embedding the loop in a cloned version of the subroutine mult yields:

```
multclone(a,b,n)
```

```
subroutine multclone(a,b,n)
integer n
real a(n), b(n)
integer i
do i = 1, n
   a(i) = a(i) * b(i)
enddo
end
```

Deciding when to embed loops and clone subroutines depends on an analysis to determine whether it is profitable to do so in the face of increasing code size [39]. Unfortunately loop embedding as presented here cannot be applied to recursive subroutines, though the benefits would be obvious. As demonstrated earlier, there is potentially significant parallelism available across partitions in divide-and-conquer algorithms. Unfortunately, the presence of recursive subroutine calls makes loop embedding pointless, exposing some (typically) constant number of partitions to parallelization rather than all the partitions. As the partition sizes (and possibly the number of partitions) vary throughout the execution of the algorithm, the amount and variance of parallelism we expose is difficult to ascertain by the compiler. We will use the following generic code example as a running example. We will elide some of the details of the transformation until after we motivate the general mechanics of embedding loops.

```
subroutine recur(...)
   Body1
do i = 1, count
   call recur()
enddo
   Body2
end
```

A simple application of loop embedding gives us:

```
subroutine recurclone(..., count_embed)
do part = 1, count_embed
   Body1
enddo
do part = 1, count_embed
   call recurclone(...,count)
enddo
do part = 1, count_embed
   Body2
enddo
end
```

The problem here should be readily apparent. We have exposed the body components *Body1* and *Body2* to parallelization within the embedded loop, but all we have done is to defer the inevitable degradation of performance due to loss of parallelism by level of the dynamic call graph. Furthermore, we now have another loop around the recursive call. Embedding this loop gives us:

```
subroutine recurclone2(..., count_embed, count_embed2)
do part = 1, count_embed2
   do part = 1, count_embed
      Body1
   enddo
enddo
do part = 1, count_embed2
   call recurclone2(...,count, count_embed)
enddo
do part = 1, count_embed2
   do part = 1, count_embed
      Body2
   enddo
enddo
end
```

Continuing in this vein, we get:

```
subroutine recurclonen(..., count_embed1, ..., count_embedn)
do partn = 1, count_embedn
   ...
      do part1 = 1, count_embed1
         Body1
      enddo
   ...
enddo
do part = 1, count_embedn
   call recurclonen(...,count, count_embed1, ..., count_embedn-1)
enddo
do partn = 1, count_embedn
   ...
      do part1 = 1, count_embed1
         Body2
      enddo
   ...
enddo
end
```

There are several problems with this code. First, the compiler cannot know where to stop embedding loops because the partition sizes are likely to be completely data dependent and

101

the problem size will probably not be known until run-time. Second, the number of clones generated here is unacceptable. This amounts to generating a different clone for each level of the dynamic call graph, a process similar to peeling off iterations of loops (loop peeling). The number of parameters being passed to each clone as well as the level of nesting increases linearly with the number of embeddings. The former problem results in increased subroutine call overhead, while the latter complicates parallelization of the loop bodies unnecessarily, increasing the cost of applying loop nest transforms and possibly inhibiting parallelizing all but the inner loop as the compiler is confronted with problems similar to those in irregular loop nests. Finally, note that for the *n*-wise embedding, *(n-2)* other clones will have to be generated (along with the original subroutine) to reach a point where the *n-1th* clone can be used.

The regularity of the structure of the clones does present the compiler with an opportunity to take steps that will collapse both the loop nests and the added parameters. A fixed point of the embedding artifacts can be constructed by explicitly counting the partitions and passing that to the embedded clone as a single parameter. The cloned subroutine need only insert one loop to surround its body components. The following clone version illustrates this:

```
subroutine recur_fixed(..., count_embed)
do part = 1, count_embed
    Body1
enddo
new_count_embed = 0
do part = 1, count_embed
    new_count_embed = new_count_embed + count(part)
enddo
call recur_fixed(...,new_count_embed)
do part = 1, count_embed
    Body2
enddo
end
```

The beauty of this simple scheme is that it limits the complexity of the resulting code while exposing all the parallelism available at each level of the dynamic call graph. In other words, the parallelism available across partitions as well as all the parallelism available in the subroutine body can be exploited. Note that the computation of the variable `new_embed_count` can be parallelized using a simple sum reduction.

We have elided the details of what happens to parameters and local variables through this transformation. In the next two subsections, we discuss the analysis and transformation of local and parameter variables to make this transformation generate semantically correct code.

6.5.1.1 Privatizing Local Variables

Local variables become subject to a unique kind of memory reuse conflict across embedded loop iterations. Since the transformation effectively emulates the execution of multiple sibling recursive calls through the use of the embedded loops, a local variable may need to be privatized for each iteration. Interprocedural anti-dependences become output dependences between iterations in the embedded loop. This may destroy values potentially needed by both the recurrent call and the subsequent phases of computation. For example, the following pattern is fairly common:

```
   subroutine recur(...)
   ...
A: a_local_variable = ...
   ...
   recursive calls
   ...
B: ... = f(a_local_variable)
   ...
   end
```

If the definition of `a_local_variable` in statement A is exposed to the use of `a_local_variable` in statement B, then a_local_variable will have to be privatized for each partition. The reasoning here is that, assuming this code has preprocessed (threshing recursive calls and eliminating return statements), the portions of code in which statements A and statement B reside will be placed in different copies of the embedded loop(s). We privatize such variables by expansion [65]. For this code example:

```
subroutine recur_embed(..., count_embed)
   ...
   do part = 1, count_embed
      ...
A:    a_local_variable(part) = ...
      ...
   enddo
   recursive calls
   do part = 1, count_embed
      ...
B:    ... = f(a_local_variable(part))
```

```
        . . .
      enddo
end
```

The key aspect of privatization, eliminating barriers to parallelization through anti-depen-
dences, is preserved in our version, though the particular application is somewhat different
than what is typical. As such, the analysis takes a different approach and is much simpler than
existing analyses for privatization.

First, we are concerned whether a variable should be privatized. Clearly, we can simply
expand all variables used in the subroutine and solve this problem, but this is not particularly
space efficient. Second, since we are applying this analysis to local variables from different
procedural contexts (or activations), no true dependences for these variables across embedded
loops iterations are possible. Thus, we need not be concerned with determining coverage of
the defs of a variable on its uses (the *dominating definitions* property) [87].

The algorithm which operates under these assumptions is extremely simple. It begins with the
DEF-USE chains for the variable. Of the three top-level clusters created by the *thresh_block*
preprocessing step, if a DEF of a variable and one of its USEs are in different clusters, then we
tag that DEF as privatized. It any DEFS of a variable are tagged as privatized, then it is priva-
tized through expansion within the iteration space of the embedded loop.

6.5.1.2 Expanding Parameters

Since the actual parameter sets for each recursive call may differ, we need a mechanism for
simultaneously passing all parameters to the embedded clone of the subroutine. We employ a
simple expansion on each formal parameter in which we pack the actual parameters prior to
the recursive call. The parameter set for each partition is accessed by the loop index variable
for the embedded loop. Note that we need not expand all the parameters, only those that differ
in the recursive calls.

The following example illustrates how it works:

```
subroutine recur(a,b,begin,end,n)
integer begin, end, n
integer a(n), b(n)
    . . .
      Body1
call recur(b,a,begin,lower,n)
call recur(b,a,upper,end,n)
```

```
     Body2
end
```

The values passed to the formal parameters `begin` and `end` differ. We expand those parameters and pack them with the parameters for each call:

```
subroutine recur_embed(a, b, begin, end, count_embed)
integer count_embed, n
integer begin(count_embed), end(count_embed)
integer a(n), b(n)
...
do part = 1, count_embed
    Body1
enddo
new_count_embed = 0
do part = 1, count_embed
    new_count_embed = new_count_embed + 2
enddo
do part = 1, count_embed
    new_begin(2*part-1) = begin(part)
    new_begin(2*part)   = upper(part)
    new_end(2*part-1)   = lower(part)
    new_end(2*part)     = lower(end)
enddo
call recur_embed(b,a,new_begin,new_end,n,new_count_embed)
do part = 1, count_embed
    Body2
enddo
end
```

In this case, we chose to consider the case of an implicit loop (whose trip count is 2). In the case of an explicit loop, the transformation is somewhat easier because the actual parameters which differ are likely to be differing array accesses into the same array. In this case, the whole array (or the portion which is used) can be passed rather than the individual reference. For example, consider the following generic subroutine:

```
subroutine recur(a,b,begin,end,n)
integer n, a(n), b(n), begin, end
...
do i = 1, count
    recur(b,a, lower(i),upper(i), n)
enddo
...
end
```

This will get transformed to the following subroutine:

```
subroutine recur_embed(a,b,begin,end,n,embed_count)
integer n, embed_count, a(n), b(n)
integer begin(embed_count), end(embed_count)
...
recur(b,a, lower(1:new_embed_count),
$      upper(1:new_embed_count),n,new_embed_count)
...
end
```

The algorithm for expanding parameters marshals together all the recursive calls and walks down each call's actual parameter list, check whether the expressions are identical. If they are identical, it does nothing. If they are not identical, it marks the formal parameter in that place for expansion. Loops are generated to pack the expanded parameter with the differing values prior to the recursive call. Note that this loop is trivially parallelizable as it includes no inhibiting dependences.

6.5.2 Embedding Conditional Statements

Embedding conditional statements is performed simultaneously with loop embedding. Unfortunately, the values used in conditional expressions for such statements are not necessarily available in subsequent recursive calls. Even if the values were available, their locations would be extremely difficult to determine in the compiler. So we simply precompute the path conditions leading to each recursive call (computed already in section 6.4.1 as *PC*) and pack those conditions into a logical array of size equal to the number of partitions. Given the set of recursive calls S, we would use the following pseudo code:

foreach $s \in S$

$$COND_s = \bigcup_{p \in PC(s)} p, \text{ where } \bigcup = \vee$$

end for

We then add the array COND as a parameter and embed the subroutine body components in if-blocks with COND as condition. Finally, we reduce the array using logical-or and use the result to guard any further recursive calls of the cloned subroutine. We use the generic example from past sections to illustrates this (with expanded parameters, expanded local variables, and embedded loop bookkeeping elided for clarity):

```
subroutine recur_fixed(..., count_embed, conds_embed)
   logical conds(count_embed), reduce_cond
   ....
```

```
do part = 1, count_embed
   if (conds_embed(part)) then
      Body1
   endif
enddo
...
do part = 1, count_embed
   new_conds(2*i-1) = conds_embed(part).and.( ⋃ₚ∈PC(part) p )

   new_conds(2*i) = conds_embed(part).and.( ⋃ₚ∈PC(part) p )
enddo
if (reduce_conds) then
   call recur_fixed(...,new_count_embed, new_conds)
endif
do part = 1, count_embed
   if (conds_embed(part)) then
      Body2
   endif
enddo
end
```

Note that we have used the high-level description of the computation of the COND array in this code, a necessity since we have not specified return conditions for this generic example. Section 6.8 will present a concrete example.

The embedded conditionals track partition liveness. When a COND array element is false, work on the corresponding partition has completed. This is detected automatically when parallelizing the code with embedded conditionals. Under the transformation as presented, parallel execution for partition will continue but no meaningful computation will take place. Unfortunately, this means that the amount of useful computation in derived parallel operations will get ever sparser as the partitions get smaller and partitions are finished. This effect will be particularly pronounced if the partitioning strategy of the algorithm results in unbalanced partitioning.

There are two strategies for dealing with this. The first is to simply ignore the problem and rely on the algorithm designer to create reasonable partitioning strategies. This is also a key consideration for the *serial* performance of the code. The second strategy is to prescan the COND array for completed partitions and pack the live partitions. This can be simply achieved by either replacing the variable count_embed with an array of those partition numbers that are live or simply compressing those expanded subroutine parameters to only include the live partitions. We then index expanded variables and parameters with elements of that

array. Prior to the recursive call to the clone, the new partition count and COND arrays are packed according to the old COND array. Another approach is to insert code to pack the arrays by the COND array at the head of the cloned subroutine.

The approach taken in our compiler is to compress the live partitions in the expanded subroutine parameters, and to reflect this in the partition count passed to the cloned subroutine. This obviates the need for an explicit COND vector, since a record of dead segments is implicitly realized by the absence of any information relevant to the execution of the algorithm on those partitions. That is, the embedded loops will only traverse structures storing information about live partitions. This may create a sparseness or irregularity in the traversal of the array structures in the algorithm; however, this is handled by the array index flattening mechanism of section 5.3.4. This results in the following code:

```
subroutine recur_fixed(..., count_embed)
    integer cond_length(count_embed)
    ...
    if (∪_{p ∈ PC(1)} p) then
        cond_length(1) = 2
    else
        cond_length(1) = 0
    endif
    do part = 2, count_embed
        if (∪_{p ∈ PC(part)} p) then
            cond_length(part) = cond_length(part-1) + 2
        else
            cond_length(part) = cond_length(part-1)
        endif
    enddo
    do part = 1, count_embed
        if (∪_{p ∈ PC(part)} p) then
            expanded_parameter1(cond_length(part)-1) = ...
            expanded_parameter2(cond_length(part)-1) = ...
            ...
        endif
    enddo
    if (cond_length(count_embed).gt.0) then
        call recur_fixed(...,new_count_embed, new_conds)
    endif
    ...
end
```

Note that this approach requires that no conditionals need to be embedded explicitly. The embedable conditionals are implicitly embedded by compressing the expanded parameter sets.

6.6 Functions

Subroutines which return values, or functions, are treated as subroutines with an extra parameter for the return value. Instances of the function name used for returning values are replaced with the new formal parameter name. Transformed in this manner, the embedding transformation presented here works as expected.

6.7 Mutual Recursion and Other Variations

This technique can easily be applied to subroutines that are mutually recursive. We have described a method to create a clone of recursive subroutines in which the control is embedded. This was achieved by adding parameters which count partitions and track embedded conditionals and by inserting loops and if-blocks which use these partitions. We can create such an embedded clone of each of the mutually recursive subroutines as if they were simply recursive, rather than mutually recursive. There is one major deviation in the mechanics of the transformations. Choosing which parameters to expand depends on the recursive calls to the subroutine, which are made in another subroutines. Otherwise, the analysis is essentially identical.

For example, consider the generic code below for a mutual recurrence:

```
subroutine recurs1(...)       subroutine recurs2(...)
    prebody                       prebody
    foreach partition             foreach partition
        call recurs2(...)             call recurs1(...)
    endfor                        endfor
    postbody                      postbody
end                           end
```

This can be transformed using the embedded clones:

```
subroutine recurs1_embed(...)        subroutine recurs2_embed(...)
    foreach partition                    foreach partition
        prebody                              prebody
    endfor                               endfor
    call recurs2_embed(...)              call recurs1_embed(...)
    foreach partition                    foreach partition
        postbody                             postbody
    endfor                               endfor
end                                  end
```

We have assumed that we can always cluster recursive calls together in preparation for embedding. However, we may only be able to create several clusters of recursive calls with other code in between. Once again, we can create an embedded clone of the subroutine, with several differences in the analysis. First, the variable privatization analysis will have to consider several more clusters, which is trivial. Second, the parameter expansion analysis will have to analyze each cluster of recursive calls and expand any parameter that needs expansion in any one of the clusters. Here is an example with two clusters containing recursive calls:

```
subroutine recurs(...)           subroutine recurs_embed(...)
    body1                            foreach partition
    foreach partition                    body1
        call recurs(...)             endfor
    endfor                           call recurs_embed(...)
    body2                            foreach partition
    foreach partition                    body2
        call recurs(...)             endfor
    endfor                           call recurs_embed(...)
    body3                            foreach partition
end                                      body3
                                     endfor
                                 end
```

The technique of creating recursively embedded subroutine clones can generally be applied across many such variations, though it is not likely that many more variations are useful to consider.

6.8 Extended Example: Quicksort

This section applies control embedding to a (stable) quicksort, a moderately complex example of a divide and conquer algorithm. The algorithm sorts an array of integer keys. It divides the problem around a chosen element, call a pivot, by creating partitions of smaller key values, equal key values, and larger key values. It then recurses on the partitions of smaller and larger keys. The sorted results are appended together in order of smallest, equivalent, and largest. (In this version, the 'append' is implicit.) The serial code follows:

```
SUBROUTINE qsort(a,b,begin,end,n)
  INTEGER begin,end,n
  INTEGER a(n), b(n)
  INTEGER lower, upper, middle, i, pivot

  IF ((end - begin) .le. 1) THEN
    RETURN
  ENDIF
  pivot = a(begin)
  upper = begin
  middle = begin
  lower = begin
  DO i = begin, end
    IF (a(i) .lt. pivot) THEN
      middle = middle + 1
      upper = upper + 1
    ELSE IF (a(i) .eq. pivot) THEN
      upper = upper + 1
    ENDif
  ENDDO
  DO i = begin,end
    IF (a(i) .lt. pivot) THEN
      b(lower) = a(i)
      lower = lower + 1
    ELSEIF (a(i) .gt. pivot) THEN
      b(upper) = a(i)
      upper = upper + 1
    ELSE
      b(middle) = a(i)
      middle = middle + 1
    ENDIF
  ENDDO
  CALL qsort(b,a,begin,lower-1,n)
  CALL qsort(b,a,middle,end,n)
  RETURN
END
```

The first steps for the compiler are to apply return eliminations, followed by threshing recursive calls. Return elimination has been applied to the version below:

```
SUBROUTINE qsort(a,b,begin,end,n)
  INTEGER n,begin,end
  INTEGER a(n),b(n)
  INTEGER lower,upper,middle,i,pivot

  IF (.NOT.(end-begin).LE.1) THEN
    pivot = a(begin)
    upper = begin
    middle = begin
    lower = begin
    DO i = begin, end, 1
      IF (a(i).LT.pivot) THEN
        middle = middle + 1
        upper = upper + 1
      ELSE IF(a(i).EQ.pivot) THEN
        upper = upper + 1
      ENDIF
    ENDDO
    DO i = begin, end, 1
      IF (a(i).LT.pivot) THEN
        b(lower) = a(i)
        lower = lower + 1
      ELSE IF(a(i).GT.pivot) THEN
        b(upper) = a(i)
        upper = upper + 1
      ELSE
        b(middle) = a(i)
        middle = middle + 1
      ENDIF
    ENDDO
    CALL qsort(b,a,begin,lower-1,n)
    CALL qsort(b,a,middle,end,n)
  ENDIF
END
```

Threshing recursive calls results in:

```
SUBROUTINE qsort(a,b,begin,end,n)
  INTEGER n,begin,end
  INTEGER a(n),b(n)
  INTEGER lower,upper,middle,i,pivot

  IF (.NOT.(end-begin).LE.1) THEN
    pivot = a(begin)
    upper = begin
    middle = begin
    lower = begin
    DO i = begin, end, 1
      IF (a(i).LT.pivot) THEN
        middle = middle + 1
        upper = upper + 1
      ELSE IF(a(i).EQ.pivot) THEN
```

```
        upper = upper + 1
      ENDIF
    ENDDO
    DO i = begin, end, 1
      IF (a(i).LT.pivot) THEN
        b(lower) = a(i)
        lower = (lower + 1)
      ELSE IF(a(i).GT.pivot) THEN
        b(upper) = a(i)
        upper = upper + 1
      ELSE
        b(middle) = a(i)
        middle = middle + 1
      ENDIF
    ENDDO
  ENDIF
  IF (.NOT.(end - begin).LE.1) THEN
    CALL qsort(b,a,begin,lower-1,n)
    CALL qsort(b,a,middle,end,n)
  ENDIF
END
```

Note that the looping structure in this case is implicit. (This is typically the case for the algorithms we have found to be useful.) Now the code is ready for control embedding. Figure 6.3 presents an annotated version of the resulting code. We have dealt with the issue of allocation in expansion by using dynamic arrays. This is convenient for code whose DCGs have static branching factors. In this case we can also pre-allocate the next DCG level's arrays. Otherwise, we employ dynamic allocation routines.

6.9 Review

We have presented a technique for embedding control in divide-and-conquer style recursive subroutines that enables effective parallelization of divide-and-conquer style algorithms. It exposes both intra- and inter-partition parallelism in the algorithm to the compiler by embedding iterative control structure. It manages the parallel execution across these partitions by implicitly embedding conditional control structure to effectively track partition liveness.

```
SUBROUTINE qsort_embed(a,b,begin,end,n,fxpartitions,
$   fxnextpartitions)
  INTEGER fxpartitions, fxnextpartitions
  INTEGER n, begin(fxpartitions), end(fxpartitions)
  INTEGER a(n), b(n)
  INTEGER lower(fxpartitions), upper
  INTEGER middle(fxpartitions), i, pivot, fxp
  INTEGER n_begin(fxnextpartitions), n_end(fxnextpartitions)

  DO fxp = 1, fxpartitions, 1
      IF (.NOT.(end(fxp) - begin(fxp)).LE.1) THEN           Embedded Loop
        pivot = a(begin(fxp))
        upper = begin(fxp)
        middle(fxp) = begin(fxp)
        lower(fxp) = begin(fxp)
        DO i = begin(fxp), end(fxp), 1
          IF ((a(i)).LT.pivot) THEN
            middle(fxp) = middle(fxp) + 1
            upper = upper + 1
          ELSE IF((a(i)).EQ.pivot) THEN
            upper = upper + 1
          ENDIF
        ENDDO
        DO i = begin(fxp), end(fxp), 1
          IF (a(i).LT.pivot) THEN
            b(lower(fxp)) = a(i)
            lower(fxp) = lower(fxp) + 1
          ELSE IF(a(i).GT.pivot) THEN
            b(upper) = a(i)
            upper = upper + 1
          ELSE
            b(middle(fxp)) = a(i)
            middle(fxp) = middle(fxp) + 1
          ENDIF
        ENDDO
      ENDIF
  ENDDO
  IF (.NOT.((end(1) - begin(1)).LE.1)) THEN
    fxplength(1) = 2                              Compressing Expanded
  ELSE                                            Parameter Through
    fxplength(1) = 0                              New Embedded
  ENDIF                                           Conditional
  DO fxp = 2, fxpartitions
    IF (.NOT.((end(fxp) - begin(fxp)).LE.1)) THEN
      fxplength(fxp) = fxcond(fxp-1) + 2
    ELSE
      fxplength(fxp) = fxcond(fxp-1)
    ENDIF
  ENDDO
  DO fxp = 1, fxpartitions, 1
    n_begin(fxplength(fxp) - 1) = begin           Partition Bookkeeping
    n_begin(fxplength(fxp)) = middle(fxp)
    n_end(fxplength(fxp) - 1) = lower(fxp) - 1
    n_end(fxplength(fxp)) = end
  ENDDO
  IF (fxplength(fxpartitions).gt.0) THEN
    CALL qsort_embed(b,a,n_begin,n_end,n,
$     fxplength(fxpartitions),fxplength(fxpartitions)*2)
  ENDIF
END
```

Figure 6.3 The resulting code for stable quicksort after control embedding.

114

Chapter 7

Compiler Architecture and Performance

This chapter briefly discusses the structure of the whole compiler. The design and ordering of compiler phases reveals much of the goals and interdependencies of each phase. This is especially true of the three major transformations we have designed in this dissertation. The earlier phases of the compiler perform relatively common analyses and transformations. The last three phases of the compiler, those new phases we have designed, must be invoked in a particular order to effect the kind of parallelization we seek. We will begin by giving an overview of our compiler, then discuss some of the standard analyses employed in the early phases of compilation, discuss some changes we have made to the dependence analysis system, and then discuss the later phases of compilation.

7.1 Compiler Overview

This thesis was implemented in the Fx compiler, a parallelizing Fortran compiler which accepts standard Fortran 77, as well as subset-HPF and task parallel extensions. This work was realized that portion of the compiler which compiled serial Fortran. The compiler generates Single Program Multiple Data (SPMD) programs for distributed memory machines and serial fortran with parallelization annotations for the Cray Research family of vector multiprocessors. Only the latter code generation scheme is fully supported by our transformations, though earlier work had support for SPMD code generation. The code generated for the Cray vector multiprocessors is then compiled by the Cray Fortran 77 Compiler.
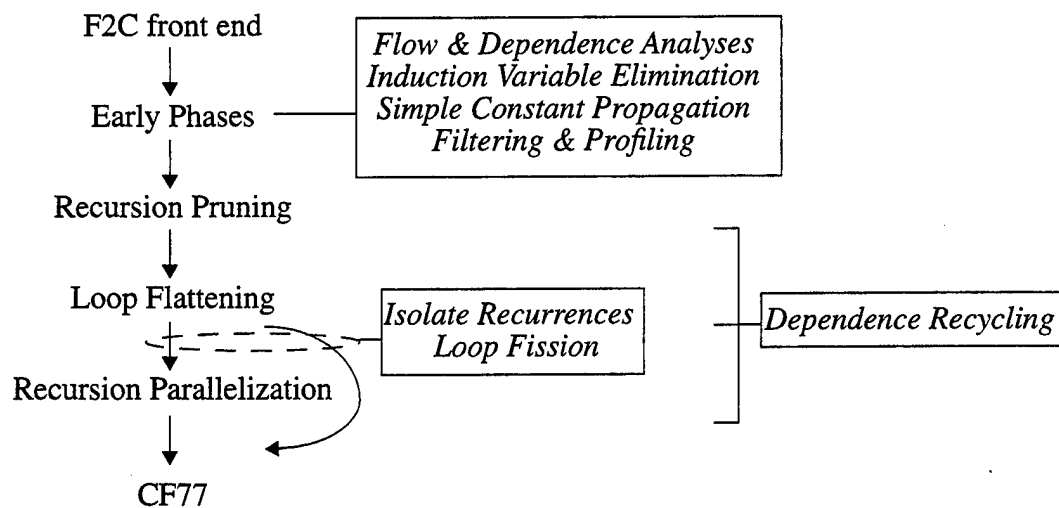
Figure 7.1 The compiler organization.

The ordering of the phases of the compiler is structured as in figure 7.1. The intermediate representation employed is an abstract syntax tree (AST) [5].

7.2 Early Passes

The compiler computes def-use chains and dependence information in the earliest phases. The dependence analyzer uses the Omega Test [69]. Simple optimizations and transformations like loop normalization, induction variable elimination, and simple constant propagation [5][93] are also implemented in these phases. The result is an AST annotated with dependence and def-use information.

7.2.1 Filtering

The compiler identifies potential candidates for transformation by using heuristics in filtering the code. Identifying irregular loop nests and recursive subroutines is relatively straightforward and simple. Irregular loops are those whose bounds are non-linear in outer loop indices. Recursive subroutines simply contain recursive calls. Those recursive routines of particular interest are those which include multiple recursive calls, each of which write to disjoint memory locations.

The difficulty is in identifying candidate recurrent loops. Simply identifying all loops with loop carried dependences is not sufficient. The compiler must decide whether within a given

```
do i = 1, N              do j = 1, M
   do j = 1, M              do i = 1, N
      y(i) = y(i) + b(j)       y(i) = y(i) + b(j)
   enddo                    enddo
enddo                    enddo
```

Figure 7.2 Loop interchanges for a loop nest.

loop nest enough parallelism can be exposed by traditional mechanism, or whether the recurrent loop itself must be parallelized. At worst, this decision can only be made at run-time or through profiling information. Figure 7.2 illustrates such a case. If N is small, then it may be worthwhile to simply parallelize the inner recurrent loops using reductions. If N is large, then it may be preferable to interchange the loops as in figure 7.2, enabling the parallelization of the outer loop.

The heuristic we employ to statically determine whether to parallelize recurrences is to simply look for loop nests that are completely recurrent, or that are nested within outer loops that have other properties inhibiting loop nest transformation. This is conservative in that it does not account for potential run-time values which might make it profitable to parallelize the recurrence loop, as seen in the previous example. However, we have not run into any cases where the wrong choice is made.

7.2.2 Profiling

Given the filtering information, the compiler can easily add profiling code. We use profiling to measure the dynamic impact of recurrences in serial code runs. Another use might be in the filtering process itself, so that run-time conditions and values are accounted for. We have not pursued this, however. Profiling directives are inserted around interesting code case. The directives used are system dependent. In the case of the Cray systems, we use the Flowtrace [2] system to profile interesting cases.

7.3 Dependence Recycling

The rough structure of each source code transformation is comprised of two phases. The IR of the code is typically examined and analyzed to determine whether and how to apply the transformations legally. The code is then transformed by either copying the code, transforming the code through substitution, or replication. For example, the loop flattening transformation

phase first examines the dependence structure of the code and the loop header structure to determine whether the transformation can be legally applied. The body components are then substituted with the precomputed loop indices and placed in the new flattened loop header.

Recomputing dependence information after each of the new transformations is not only extremely costly, but, more importantly, it might result in a loss of accuracy in the dependence information due to the introduction of indirection in array accesses and complex expression computed in subroutine calls. However, in all but the recurrence parallelization phase, the resulting dependence graph will be isomorphic to the original dependence graph. The reason is that we essentially preserve the data flow semantics of the code by only manipulating loop control structure in a very restricted manner during the second phase of these transformations. That is, though depth of loop nesting may be increased or decreased through the transformations presented here, the flow constraints on the ordering of statement execution is not violated. We thus preserve the original dependences through the transformations since the effective structure of the dependence graph is unchanged. The only thing that may change are the distance (or distance) vectors, especially in the case of loop flattening.

The mapping is implemented in a relatively simple and non-intrusive manner by instrumenting the copying routines in the compiler source to automatically maintain and update a mapping between the original code and transformed code. When a piece of code at one end of a dependence link is mapped, the link is either copied or it is updated to reflect the transformed code. The first time a dependence link is encountered, it is copied and the unresolved end of the link is entered into a mapping table by its old destination address. If the link has already been copied, determined by lookup in the mapping table, the end of the link pointing to the address being copied is updated to reflect the new address. Note that for a given address, multiple links may be updated or copied depending on the order of the copy operation and the structure of dependences in the code.

The code for the transformations is barely changed at all. The basic statement and expression copying and substitution are modified to perform the dependence mapping automatically. The only thing added to each transformation phase is a subroutine call to turn the mapping process on and off around copying phases.

7.4 New Passes

The interdependencies between the new passes are obvious. Since it nests control structure which might (and probably will) result in loop nests, the recursion pruning pass must precede the loop flattening phase. Both the loop flattening and recursion pruning phases generate recurrent loops, so they must precede the recurrent loop parallelization pass.

The loop flattening pass works on a per loop nest basis, while control embedding works on entire subroutines. The recurrent loop parallelization pass is comprised of two phases. The first isolates recurrent portions of loops through loop fission, and the second performs the actual analysis. The rationale for this decision is to make the parallelization process clean, so that unnecessary computation is not embedded within the templates for performing the parallel recurrent primitives. Furthermore, different recurrent components may need to utilize different templates. The drawback, however, is that this may lead to inefficient usage of these parallel templates. Two reductions in one loop may benefit by sharing the overhead of performing the reduction. However, these inefficiencies can be mitigated by fusing together the generated recurrent primitives.

The phase which isolates recurrent portions of loop bodies starts by finding strongly connected components of flow, output and control dependences. The compiler then performs loop fission to distribute loops around recurrent sections of code. This entails promoting scalar values to arrays if they are computed in one loop and used in another. The recurrent loop parallelization phase then parallelizes those loops that are recurrent.

7.5 Tracking Recurrent Primitives

The type of recurrent primitive applicable to a recurrent loop may change through the control structure transformations presented here. For example, consider a recurrent loop which computes a reduction in a hypothetical recursive subroutine. Control embedding will embed that loop in another loop. The transformed code will now include indirect array accesses due to promotion of variables during control embedding, the compiler would have to resort to a more costly combining-send or multiprefix operation. However, by keeping track of the original form of the loop, the compiler can employ a less costly segmented reduction. Furthermore, if intermediate results of the computation are used, e.g. in a permutation of an array, then the type of primitive employed should be a segmented scan. This is determined in the loop fission

119

phase, when recurrences are isolated and extracted from non-recurrent code which may use intermediate values computed by the recurrent code. The transformation of the particular recurrent primitives used to compute the code can be specified by the following tables. The first table specifies recurrent primitive transformation through loop flattening The type of resulting primitive depends whether the inner index variable(s) of the flattened loop nest occurs:

Source Primitive	Resulting Primitive Through Loop Flattening	
	No Inner Index Occurrence	Inner Index Occurrence
reduction	segmented reduction	combining-send
scan	segmented scan	multiprefix
combining-send	combining-send	combining-send
multiprefix	other	other

The next table specifies the transformation of the primitive types through the loop fission phase. The resulting primitive may change if flow dependences originating in the cluster of statements which compute the primitive crosses the cluster boundary:

Source Primitive	Resulting Primitive Through Loop Fission	
	Flow Deps Cross Clusters	No Flow Deps Cross Clusters
reduction	scan	reduction
segmented reduction	segmented scan	segmented reduction
scan	scan	scan
segmented scan	segmented scan	segmented scan
combining-send	multiprefix	combining-send
multiprefix	multiprefix	multiprefix

120

7.6 Computation and Space Overhead Reuse

Flattened loops and some recurrent primitives have large overheads in building and storing data structures which may be reusable. For example, the index computation and arrays for a flattened loop can be reused for subsequent flattened loops with similar index iterations spaces. Likewise, a combining-send operation builds the SPINE structure, which can be reused in other combining-send and multiprefix operations with identical index arrays and array sizes. Reuse of such computation and allocated space is crucial to achieving reasonable performance. A simple framework for general computation and space reuse is built into the compiler.

The space reuse problem can be framed in a manner similar to the register allocation problem. The primary difference is that space reuse is restricted by the type and size of the allocated object. Conceptually, this effectively divides up the candidate memory blocks into disparate pools in which the reuse algorithm is run separately. To an extent, tricks can be played to avoid this division, though the size restrictions are useful to retain to avoid overallocation of space for the sake of increasing the occurrence of size affinity in the reuse algorithm.

The computation reuse problem can be framed similarly. Live ranges of the computed values for a recurrent primitive or a flattened loop can be simply inferred as the extent of the trans-formed code. The primary reason for this is that is that the values computed in the setup phases of the transformed code are likely to be used throughout the execution of that code. Despite the apparent conservatism of this approach, it is unlikely that flattened loops or recurrent primitives will overlap in way such that opportunities for reuse will be lost under this assumption.

7.7 Compiler Performance

Compile speed of the passes described this dissertation is relatively fast, despite being unoptimized. Compilation of large programs is dominated in both speed and time by the dependence analyzer for large programs. Loop flattening and recursion pruning take negligible amounts of

time. Recurrence parallelization is somewhat more costly, taking between 5 and 12 seconds per loop on a Sparcstation ELC, a relatively slow workstation compared to more recent workstations:

Loop	Analysis Time (sec.)
Linear	4.9
Max	9.1
MaxSub	11.5
(Failure)	13.4

In the last row of this the table is a case where the analysis fails to find an efficient composition operator. Note that in this case, the number of iterations is bounded. If we increase the ceiling on the number of iterations through the analysis, this number would be significantly higher.

However, the performance of the recurrence parallelizer is severely unoptimized, the most prominent cost being the start-up time for the module responsible for this pass. It is restarted for every recurrent loop and takes roughly 8 seconds to start up.

7.8 Review

This chapter discussed the overall organization of the compiler. Important techniques like space and computation reuse and dependence recycling form the 'glue' which allow the transformations discussed in prior chapters to work together and generate efficient code. Recognition and transformation of recurrent primitive types through transformations provide valuable hints to later phases of the compiler to generate more efficient code. The compiler also relies on a base of more standard optimizations and analyses to decide when to apply those transformations. Finally, measurement of the compile time are given and shown to be reasonable.

Chapter 8

Evaluation

The evaluation of the compiler transformations in this thesis takes several paths. For the base recurrence parallelization technique, a simple measure is to compare the number of loops parallelized again other compilers, such as the summary of compilation results on the Argonne loops in the introduction. Accordingly, we also note here which strategies our compiler and the CF77 employs to parallelize components of the test programs, which provides a bridge between per loop compilation performance and overall program performance. Another important measurement to consider for later performance estimates is the performance of the code templates used for parallel recurrent primitives. Since the focus of this thesis is the compiler rather than the intricacies of engineering efficient parallel primitives, measurements of the primitive performance serve mainly as a useful calibration guide when examining their performance in the larger context of whole algorithms or programs. Finally, we examine the impact of the transformations on various algorithms.

Unless otherwise noted, the performance numbers presented here are for a single vector processor of a Cray C90 supercomputer[1]. The performance of our compiler is typically denoted in graphs by the word 'auto', while the performance of the Cray Fortran 77 parallelizing compiler is denoted by the acronym 'CF77'.

---

1. To quote the Pittsburgh Supercomputing Center's Guide to Supercomputing:

"The CRAY C90-16/512 has 16 processors with a peak aggregate speed of 16 Gflops and a main memory of 512 MWords or 4 GBytes. "

8.1 Overview

Generally, for each array length multiple arrays were generated randomly and used in each test. The results presented are comprised of the arithmetic mean of these runs. In cases were appropriate, other testable parameters were varied and tested in the same manner. For example, the key density of the arrays used in combining-send or multiprefix operations was varied to assess the sensitivity of the primitive to such variations. The performance numbers presented for each program typically includes both a computational throughput measure, if a meaningful one exists, as well as a relative speedup graph, which charts the speedup of code that we generate relative to code that the CF77 compiler generates.

The first set of programs presented are essentially simple loop kernels. The relative performance of these kernels as compiled by the various compilers should give an expectation of how well the basic recurrence parallelization technique performs in more complex contexts. The second set of programs represent higher-level algorithms which use both the recurrence analysis technique, as well as the controls structure transformations to various degrees. In addition to performance graphs, the particular parallel primitive and control structure transformed is indicated for each program.

8.1.1 Other Performance Factors

The array lengths used were generally sampled uniformly in increments non-integral in the vector register length of the machine. The primitive deployed by the compiler picks shape factors for the computation to avoid memory bank conflict. In particular, strided memory accesses of multiples of 4 are avoided if possible. Thus, the average vector length of the operations used typically fluctuates slightly below the vector register length of the machine. The fluctuations in the performance graphs generally correlate to the average vector length.

8.2 Code Template Performance

The code templates described in Chapter 2, along with the derived modeling functions and composition operators, provide the basis for generating parallel code for the recurrence parallelization phase. The code templates we use are those for reductions, scans, combining-sends, and multiprefix. Evaluating the performance of these templates is critical to the overall evaluation of code parallelized by our compiler.

124
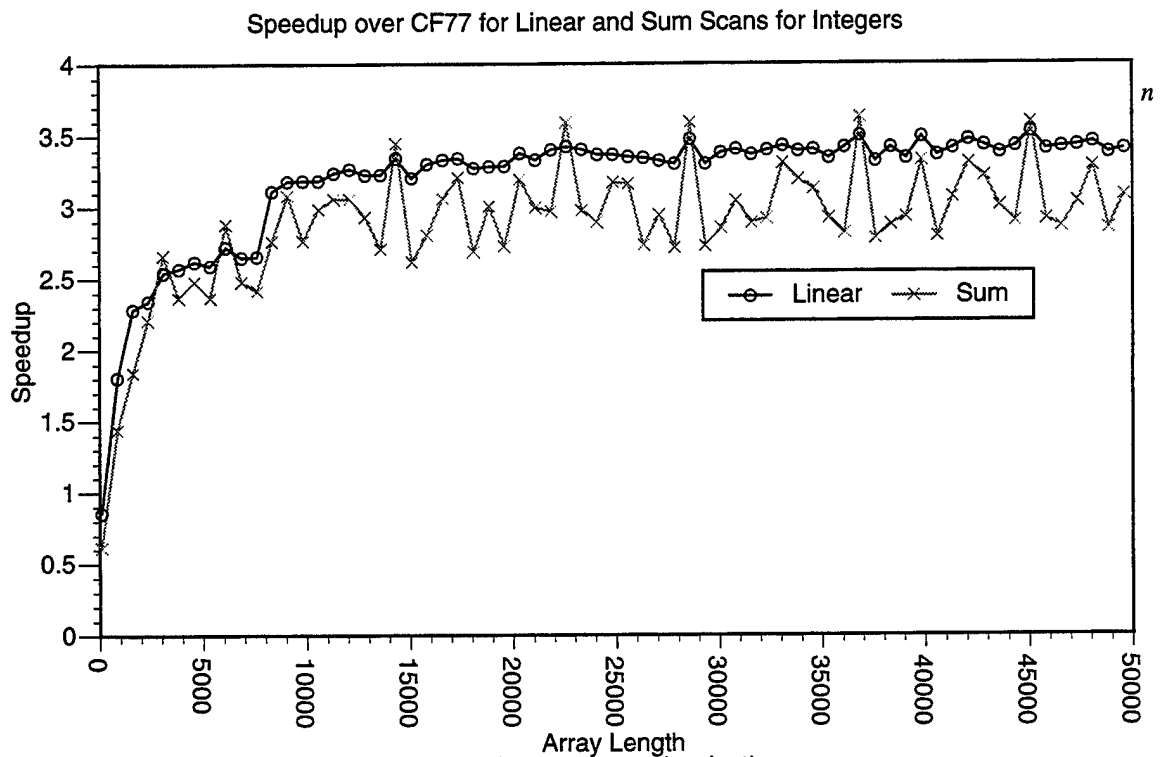
Figure 8.1 Relative speedups for integer linear scan and reductions.
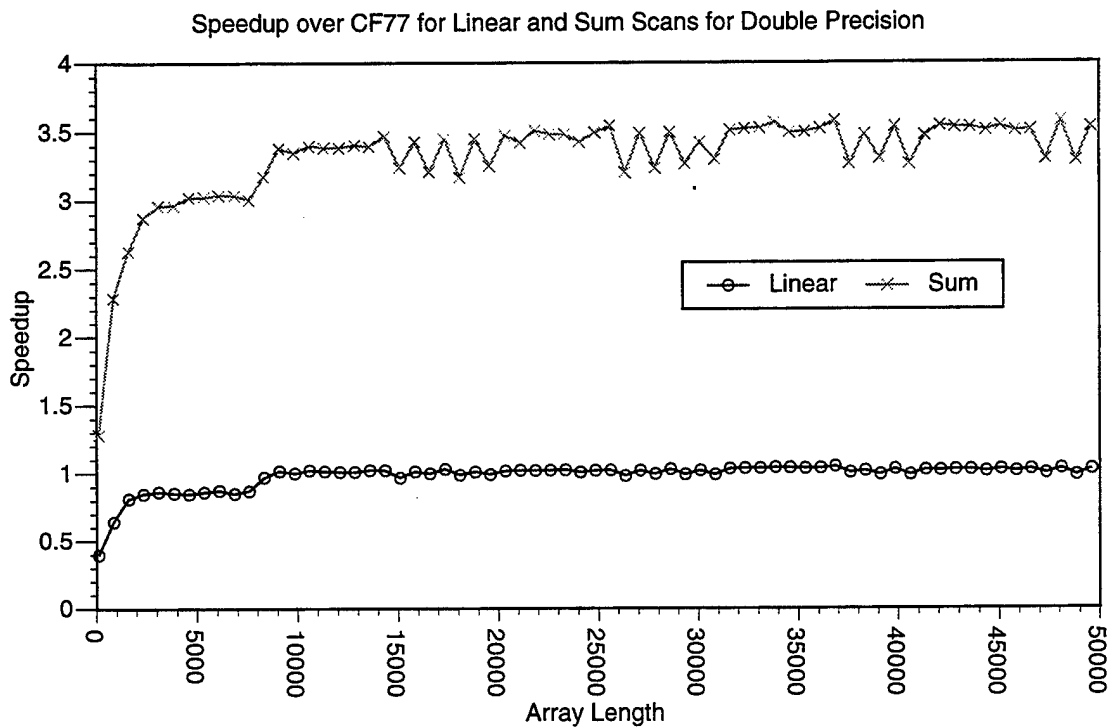
Figure 8.2 Relative speedups for double precision linear scan and reductions.

While the code templates were not highly optimized, the best available algorithms were selected and implemented based on the following criteria: performance, operator generality (i.e. associativity is sufficient), and mappability to vector multiprocessors. For the purposes of evaluating their performance, the templates were each instantiated with several composition

125

operators. In the cases of combining-send and multiprefix operations, performance at various key densities were sampled.

The two modeling functions used for measuring reduction and scan template performance were a sum and linear recurrence. The relative speedup over the Cray Fortran 77 Compiler (CF77) is plotted in figures 8.1 and 8.2. The speedups over CF77mostly range between 2.5 and 8 for reasonably large array sizes. The sole exception to this is for linear scans of floating point numbers and simple sum reductions. In this case, the user can enable pattern recognition of linear recurrences through a command-line flag to CF77. The routine invoked (*folr2p*) [1] is a highly optimized assembly-coded routine designed to solve linear recurrences on a single head of the C90. In instances where this is invoked, the performance of our code lags behind that generated by CF77. However, note that this routine is limited to a single vector processor, whereas our template scales to multiple heads. Our template is not particularly optimized, as well as being written in Fortran. Finally, our template supports a more general class of recurrences, rather than just linear recurrences.

This points out a general disparity between the performance of our templates and the CF77 generated code for reductions and scans. CF77 resorts to highly optimized code when it successfully pattern matches. Otherwise, it must resort to serial code. Our compiler generates code template in which little effort has been applied at tuning and optimization. Nevertheless, the performance are reasonably close in cases where CF77 resorts to optimized library routines, and beats it handily otherwise. However, nothing precludes our compiler from generating templates that have been hand-tuned. Results presented later in this section reveal that the presence of conditional branching some loops causes larger disparities in performance because of the difficulty optimizing conditional branches in the CF77 compiler's serial optimizer.

The two modeling functions used for measuring combining-send and multiprefix templates are a simple scalar increment (i.e. histogram) and an articulated maximum (maximum expressed through conditional statement, rather than intrinsic subroutines). The latter is more representative of the general case, as CF77 (and most other compilers) is unable to parallelize general combining-send or multiprefix operations.The relative speedup over CF77 is plotted in figures 8.3, 8.4, 8.5, and 8.6. Multiple key densities are samples and plotted in each figure. The variance due to key density is relatively small, with a general trend toward better relative
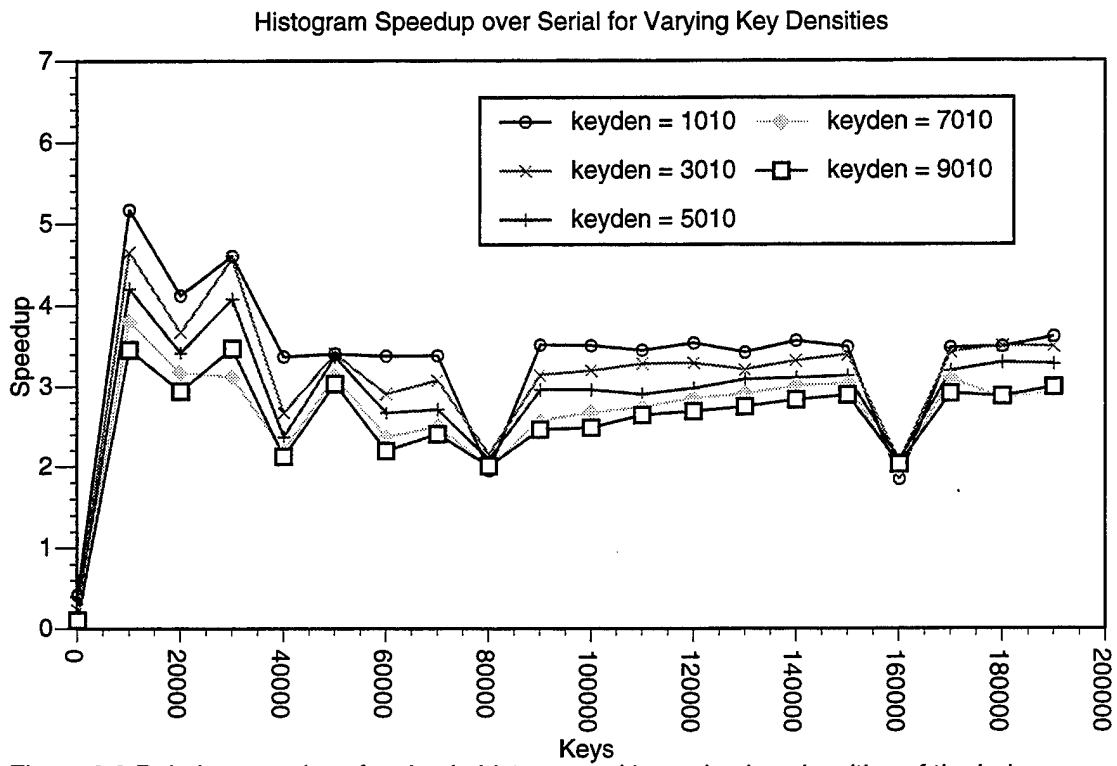
Histogram Speedup over Serial for Varying Key Densities

Figure 8.3 Relative speedups for simple histogram with varying key densities of the index array.

Multiprefix Speedup over Serial for Varying Key Densities
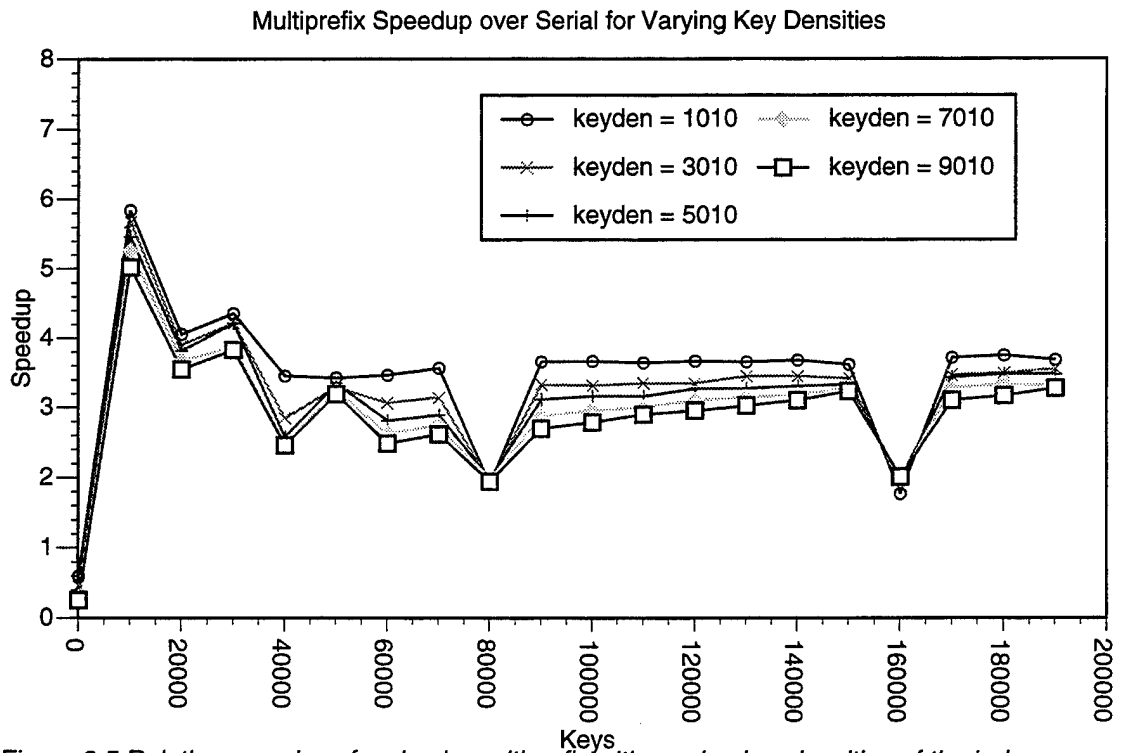
Figure 8.5 Relative speedups for simple multiprefix with varying key densities of the index array.

General Histogram Speedup over Serial for Varying Key Densities

Figure 8.4 Relative speedups for generalized (max operator) histogram with varying key densities of the index array.
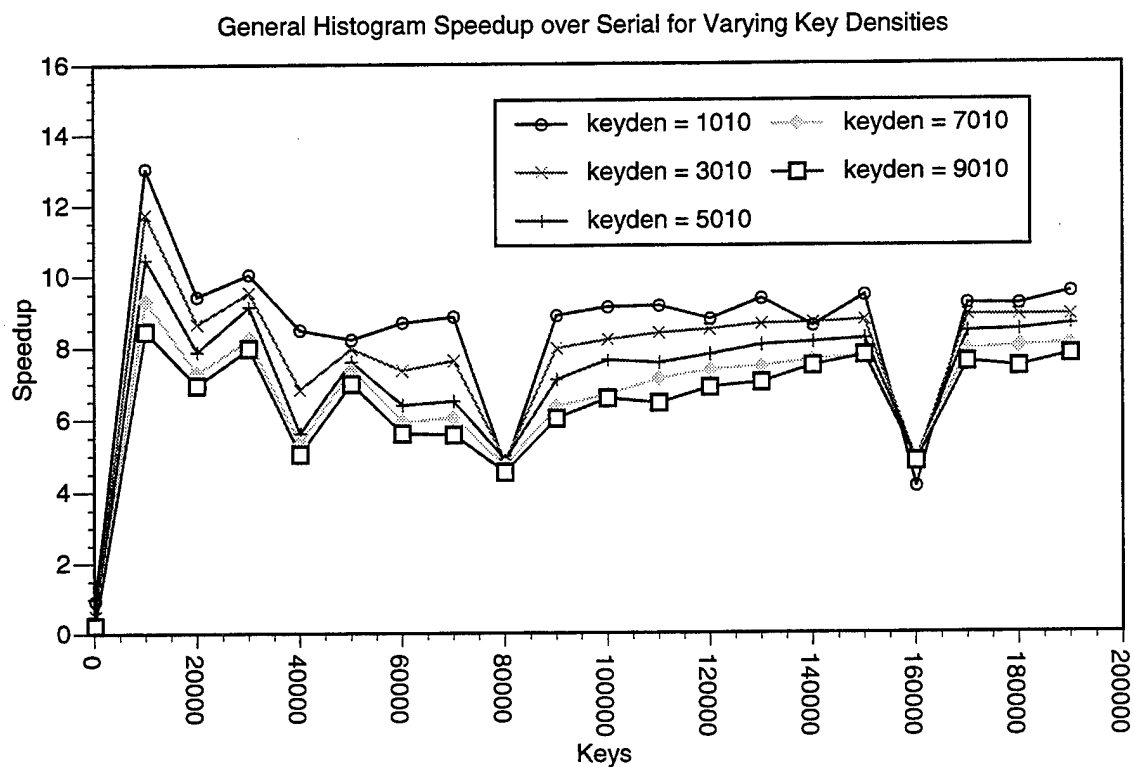
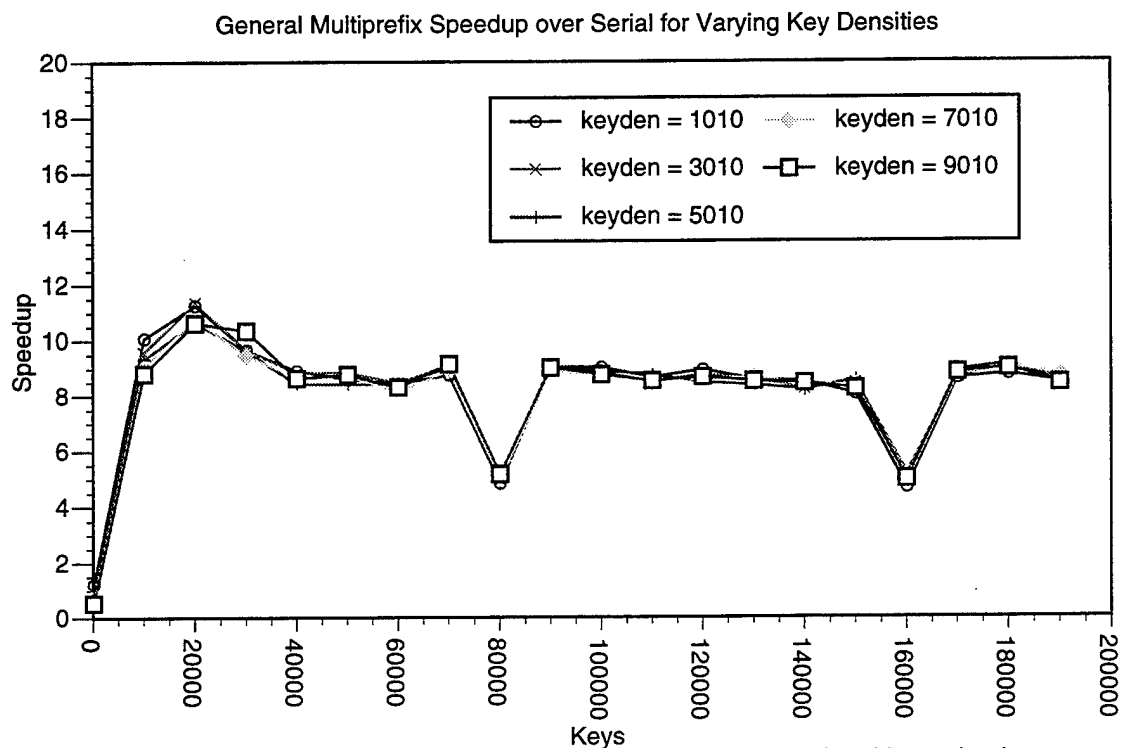### General Multiprefix Speedup over Serial for Varying Key Densities

Figure 8.6 Relative speedups for generalized (max operator) multiprefix with varying key densities of the index array.

performance with lower key densities. There are several reasons for this. Higher key densities increase the storage requirements in the primitive template, and the associated initialization costs. Furthermore, the performance of the CF77 version, which, in some cases, is no better than that of the serial version, may degrade with lower key densities due to increased serialization of memory accesses. The overall performance improvement over CF77 varies a bit more than for reductions and scans, ranging from 2 to 14.

8.3 Compiler Passes in Use

The particular primitive or control structure deployed by our compiler for the programs is denoted in the table below. There are four possible primitives generated: Reduction (RD), scan (SC), combining-send (CS), and multiprefix (MP). The two control structure transformations loop flattening (LF) and control embedding (CE) are also denoted where they are used. Finally, the table includes information about the CF77 compiler success at parallelization. We denote cases were that compiler could not parallelize any of the code as 'Scalar', some of the code as 'Partial', and all of the code as 'Full':

Benchmark	Parallel Primitive				LF	CE	CF77
	RD	SC	CS	MP			
Maxsubsequence	X	X					Scalar
Partition		X					Scalar
Segpartition		X			X		Scalar
Bucketsort		X	X	X			Partial
CSR Spmatmul	X				X		Full
CSC Spmatmul			X		X		Full
Simple Quicksort		X			X	X	Partial
Stable Quicksort	X	X			X	X	Partial
Quickhull	X	X			X	X	Partial

The important cases to note here are those for which the CF77 compiler can fully parallelize. The CSR and CSC sparse matrix-vector multiplication kernels can both be parallelized by the CF77 compiler. The difference is that our compiler is able to automatically derive a better algorithm (really, a primitive) in parallelizing the algorithm, as the performance results presented later in this chapter will indicate.
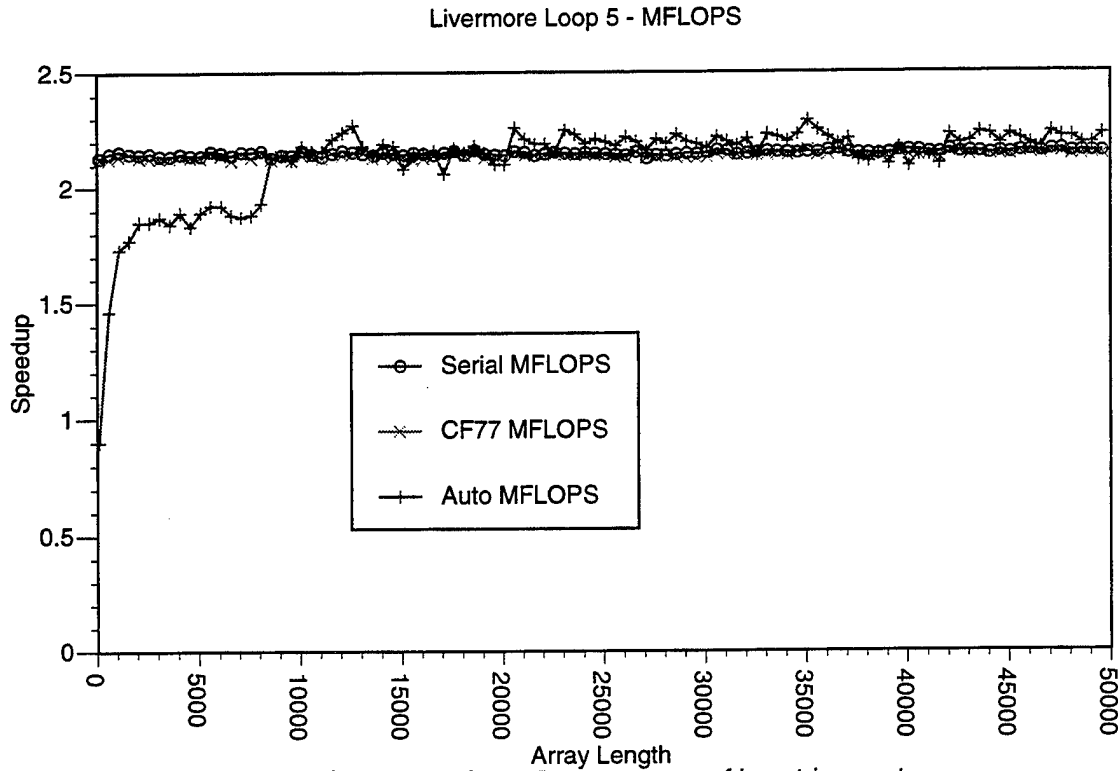
Livermore Loop 5 - MFLOPS

Figure 8.7 Performance of Livermore Loop 5 over a range of loop trip counts.

8.4 Algorithms Parallelized

From the perspective of the user, the most important measurement is the performance improvement in their applications. Furthermore, the impact of the control flow transformations built on top of the recurrence parallelization is best measured in applications which utilize irregular control flow structure. This section discusses algorithms which utilize all of the recurrent primitives and control structures discussed in this thesis.

8.4.1 Livermore Loops

There are three kernels of interest to us in the Livermore Loop suite [34]. In particular they are loops 5, 19, and 24, all recurrent loops. Loops 5 and 19 are both linear recurrences. Loop 19 is comprised of two similar recurrent loops.

```
    Do 5 i = 2,n
5      X(i) = Z(i) * (Y(i) - X(i-1))



    do 191 k = 1,n
       B5(k) = SA(k) + STB5 * SB(k)
191    STB5 = B5(k) - STB5
```

130

Livermore Loop 19 - MFLOPS
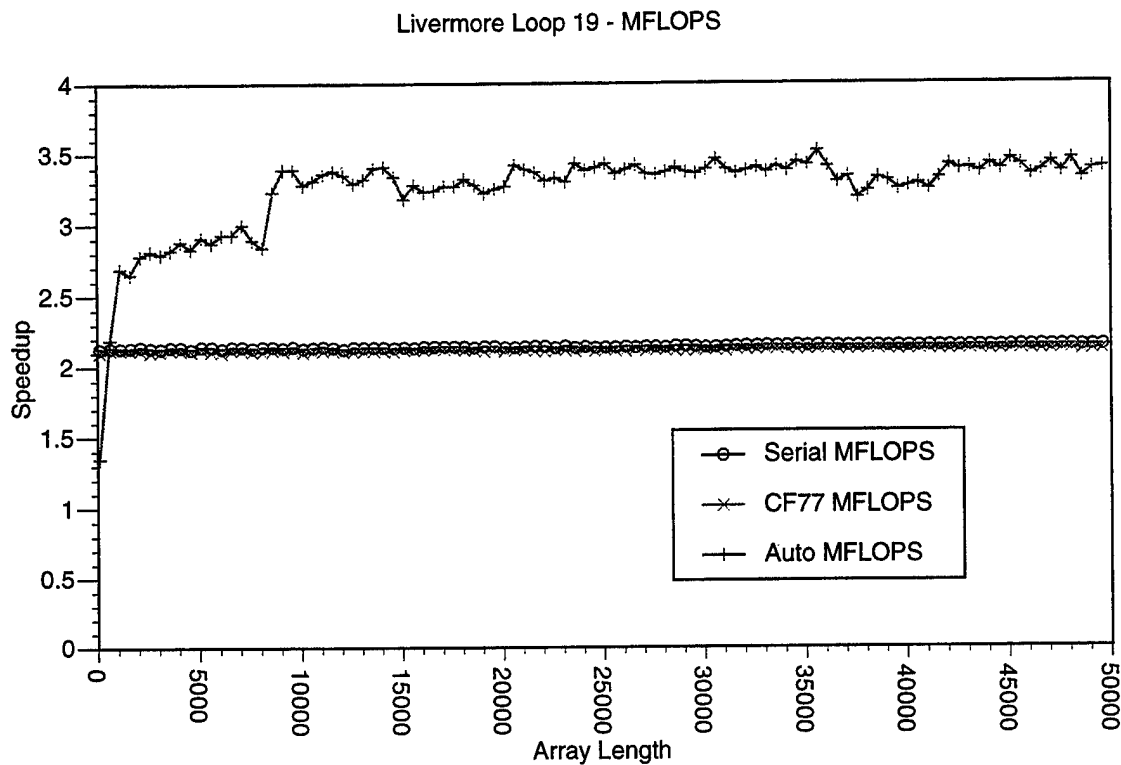
Figure 8.8 Performance of Livermore Loop 19 over a range of loop trip counts.

```
    do 193 i = 1,n
        k = n - i + 1
        B5(k) = SA(k) + STB5 * SB(k)
193     STB5 = B5(k) - STB5
```

Loop 24 locates the index of the first instance of the maximum value in an integer array:

```
    do 24 k = 2,n
24      if(X(k).lt. X(max24)) max24 = k
```

The relative performance and speedups are plotted in figures 8.8, 8.7, and 8.9. The one kernel which obviously suffers in performance is Loop 5. The primary reason for this is the overhead of instantiating the template variables for the derived scan operation. This overhead can be mitigated by fusing the loops in which template variables are instantiated with the loops of the scan primitive.

131

Livermore Recurrences - Speedups Over CF77
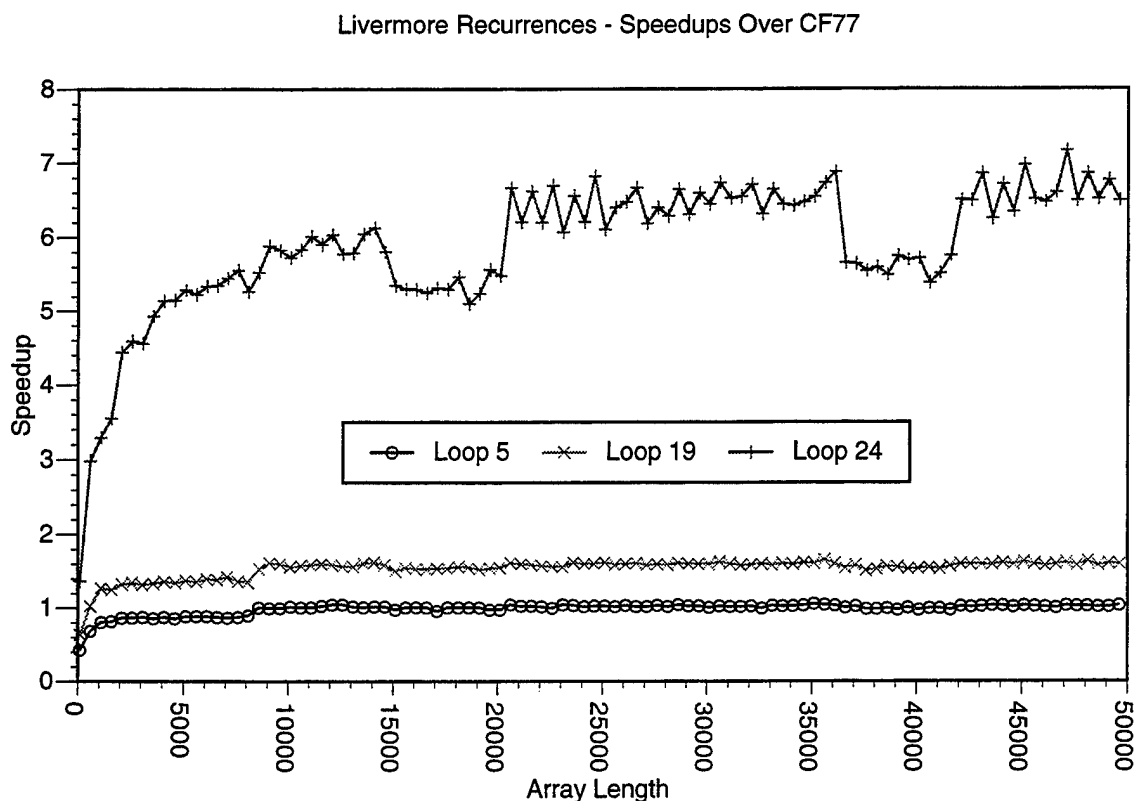
Figure 8.9 Relative speedup of selected Livermore loop suite recurrences over a range of loop trip counts.

8.4.2 Maximum Subsequence Sum

This kernel computes the largest non-zero sum of contiguous subsequence sum of a series of numbers [13]. The difficulty in parallelizing this recurrent loop is that the operator used in a reduction or scan is fairly different from the serial loop. So, no associative operator is obvious from an inspection of the code:

```
integer a(n)
integer i, sofar, max

do i = 1, n
   if (sofar + a(i) .lt. 0) then
      sofar = a(i)
   else
      sofar = sofar + a(i)
   endif
   if (max .lt. sofar) then
      max = sofar
   endif
enddo
```

Speedup over CF77 for Maxsubsequence

Figure 8.10 Relative speedup of maximum subsequence sum kernel.

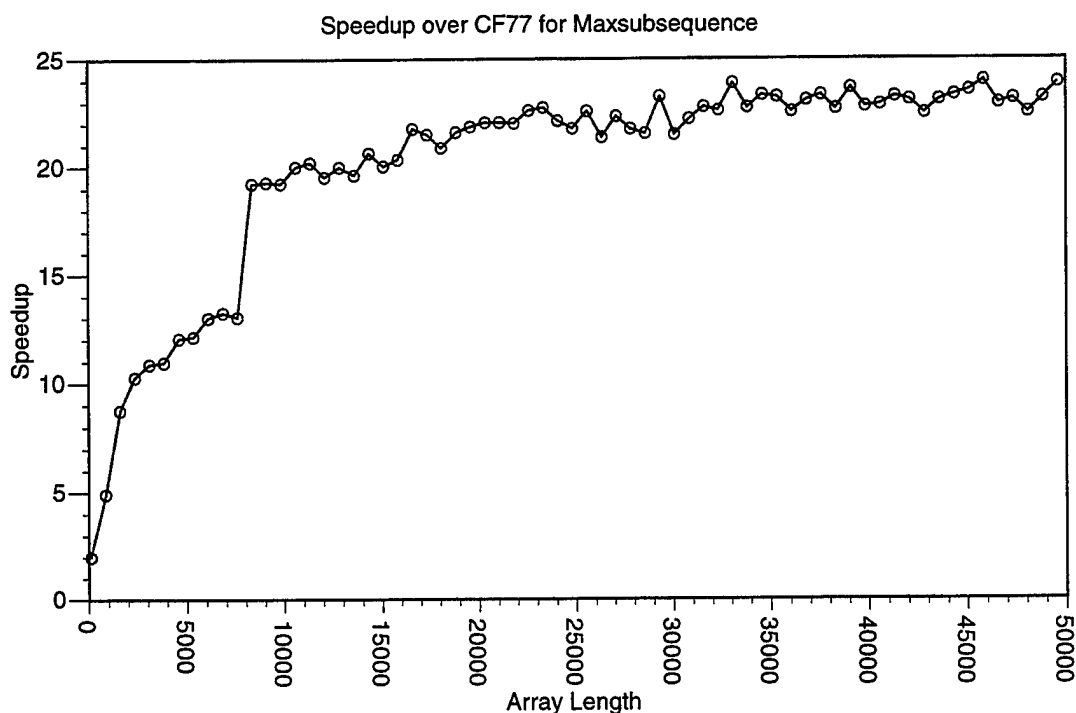The CF77 compiler cannot parallelize this kernel, so the performance measured is for serial code. The presence of two conditional branches in this code results in poor scalar performance for the code, so the resulting performance disparity is high. The relative performance is plotted in figure 8.10. The derived composition operator for this kernel was discussed in detail in section 4.2.

8.4.3 Sparse Vector-Matrix Multiplication

The CSR and CSC kernels were compiled for both a single vector processor and 4 vector processors of the C90. In compiling for a single head, the CF77 compiler vectorizes the reduction in the inner loop of CSR. In compiling for multiple heads, the CF77 compiler tasks the outer loop of the kernel and vectorizes the inner loop reduction of CSR. For this kernel, our compiler generates a segmented reduction, whose pseudo-Fortran is below, which is simultaneously tasked and vectorized:

```
flat = pntr(N+1) - 1
    vecwork(1:flat)  = val(1:flat:1) *
    $                         vec(indx(1:flat))

condition = k(1:flat) .le. pntr(i(1:flat)+1) - 1
```

133

```
y(1:N)  = APPLY(FUN_REDUCE(
     $                {λx.(condition?(x + vecwork):vecwork)},
     $                flatlength,
     $                (i(1:flat - 1) .ne. i(2:pntr(N+1)-1))),
     $          Y(1:N))
```

The brackets on the lambda expressions denote the analyzed version of the enclosed loop modeling function. The third argument to the function composition reduction (FUN_REDUCE) is simply shorthand for a pack of the results. The CF77 compiler only vectorizes the inner loop of the CSC compiler on both single and multiple head configurations. Our compiler generates a combining-send operations which is simultaneously tasked and vectorized, below:

```
vecwork(1:flat)  = val(1:flat:1) *
     $                        vec(i(1:flat))

y(1:N)  = APPLY(FUN_COMB_SEND(
     $                {λx.x + vecwork},
     $                i(1:flat)
     $                flat,
     $                N),
     $          Y(1:N))
```

Sparse matrices were generated randomly, varying the average row lengths and number of rows. The relative speedup over the Cray compiler for CSR is plotted in figure 8.15 for roughly 100,000 elements, varying the average row length. Our compiled CSR on a single processor is faster than the Cray version up to an average row length of 110, with a peak relative speedup of 13.5. On four processors, the peak relative speedup for these matrices is over 6, with a crossover at an average row length of 85. Figures 8.12 and 8.11 plot MFLOPS against the average row length for our technique, the Cray CF77 compiler, and the optimized assembly level routine [20], which we refer to as SEGMULV. Note that the peak performance reported here for this library is slightly lower than reported in the source reference due to the differing C90 configurations. For both single and multiple processors, our technique and the library routine sustain a relatively flat MFLOPS rate across varying average row lengths. The CF77 compiler appears to depend on the average row length, as one would expect given that this constrains average vector length for the compiled kernel. There is still a good deal of optimization that can be performed on our code templates, as evidence by the constant overhead between our technique and the library's performance. We will discuss this further below.

MFLOPS for CSR Sparse Matrix-Vector Multiplication on 1 Processor
(100k non-zero elements)

Figure 8.11 Performance of CSR sparse matrix-vector multiplication kernel for a single vector processor.

MFLOPS for CSR Sparse Matrix-Vector Multiplication on 4 Processors
(100k non-zero elements)
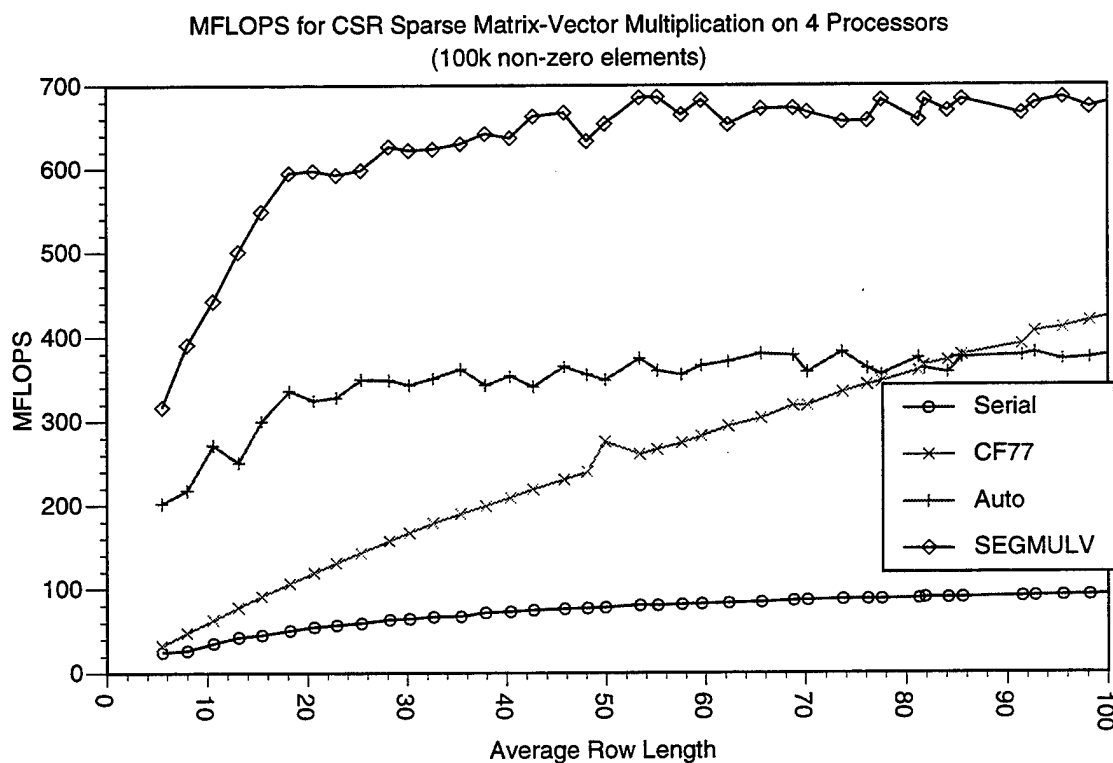
Figure 8.12 Performance of CSR sparse matrix-vector multiplication kernel for four vector processors.

135

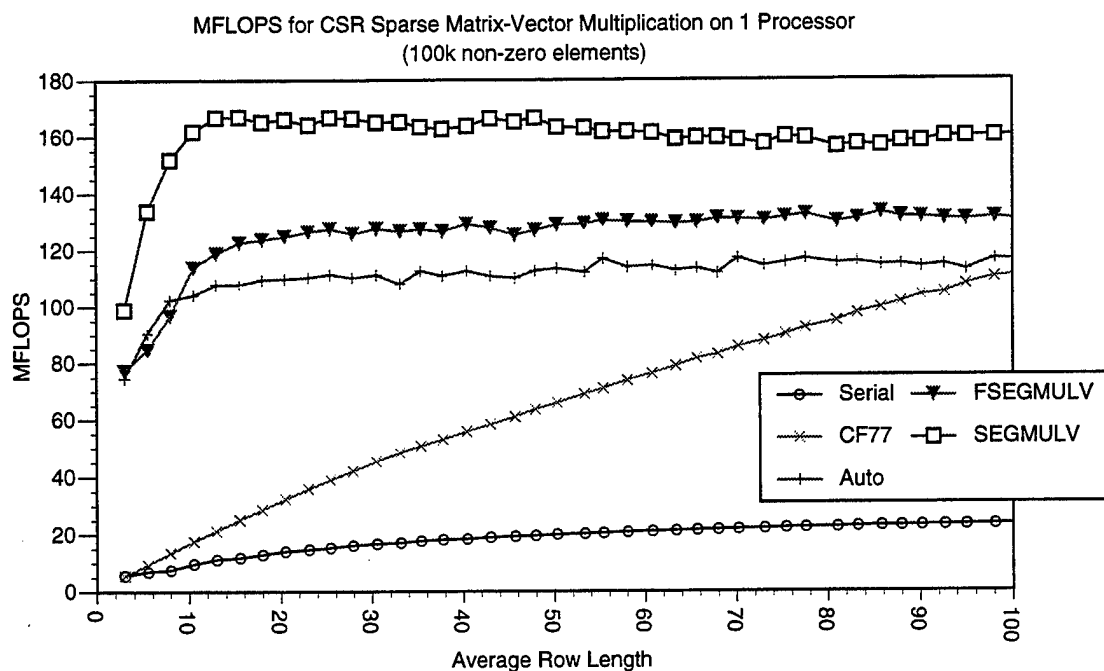**MFLOPS for CSC Sparse Matrix-Vector Multiplication on 1 Processor**
(100k non-zero elements)
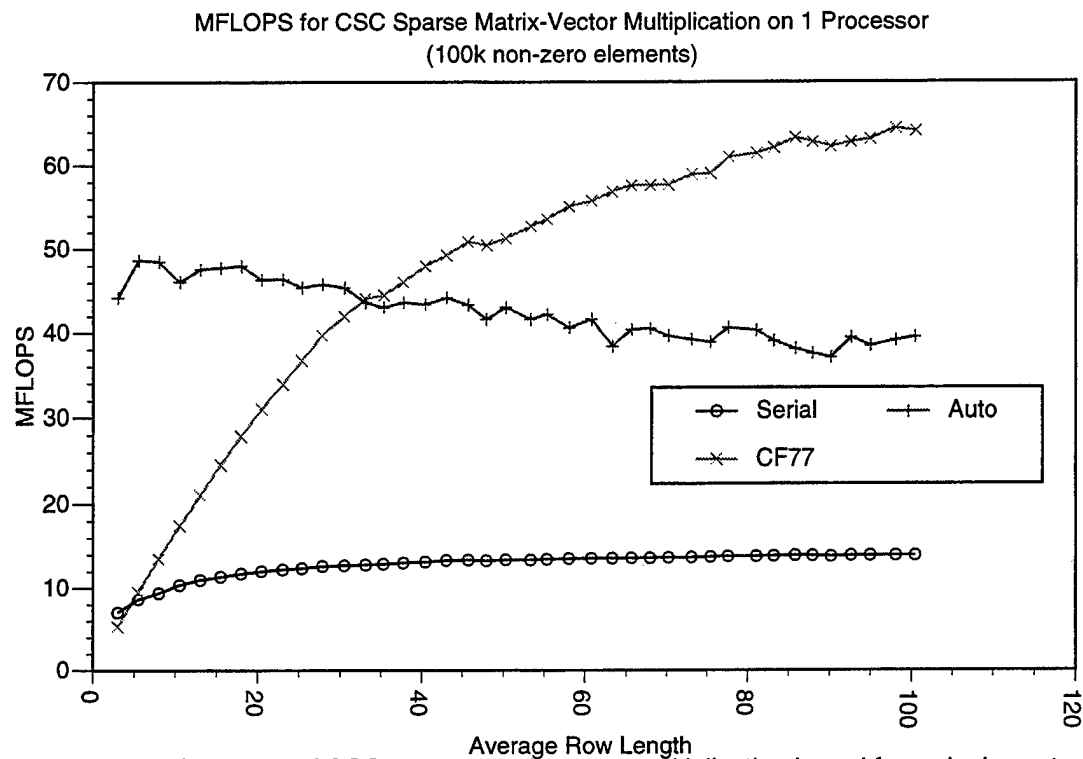
Figure 8.13 Performance of CSC sparse matrix-vector multiplication kernel for a single vector processor.

MFLOPS for CSC Sparse Matrix-Vector Multiplication on 4 Processors
(100k non-zero elements)
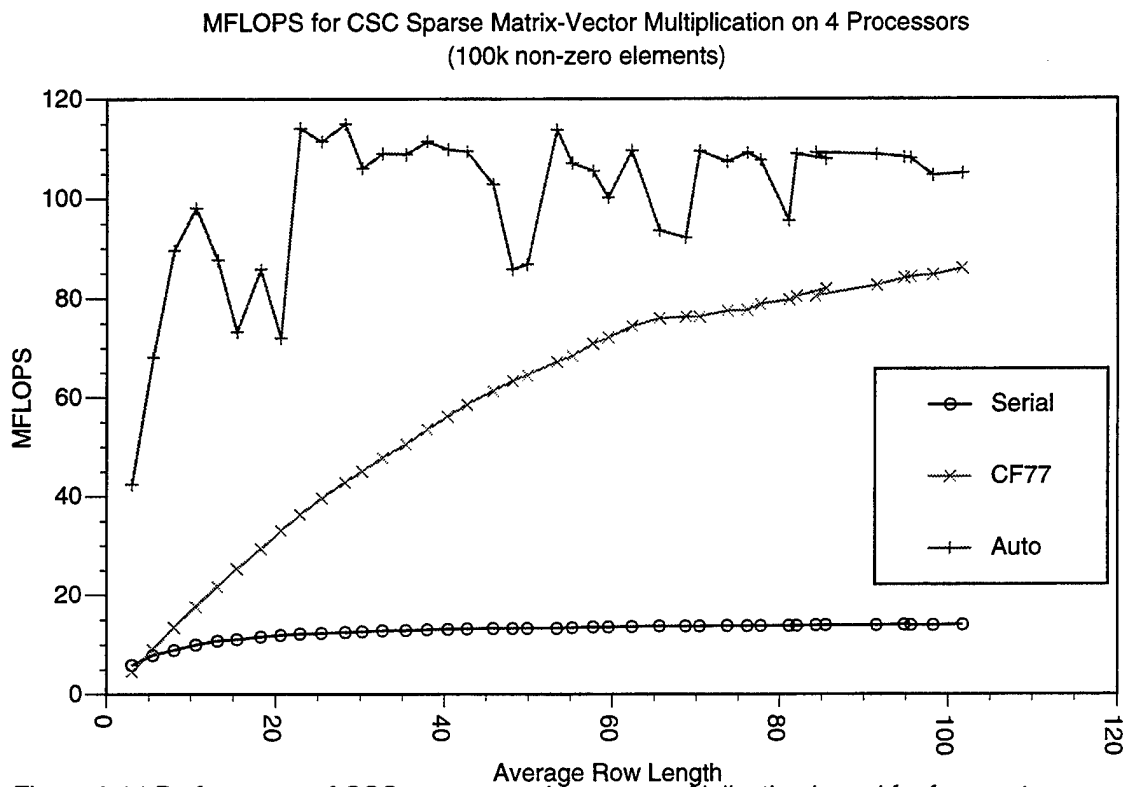
Figure 8.14 Performance of CSC sparse matrix-vector multiplication kernel for four vector processors.

136

Speedup of Loop Flattening over CF77 for CSR Sparse Matrix-Vector
Multiplication (100k non-zero elements)
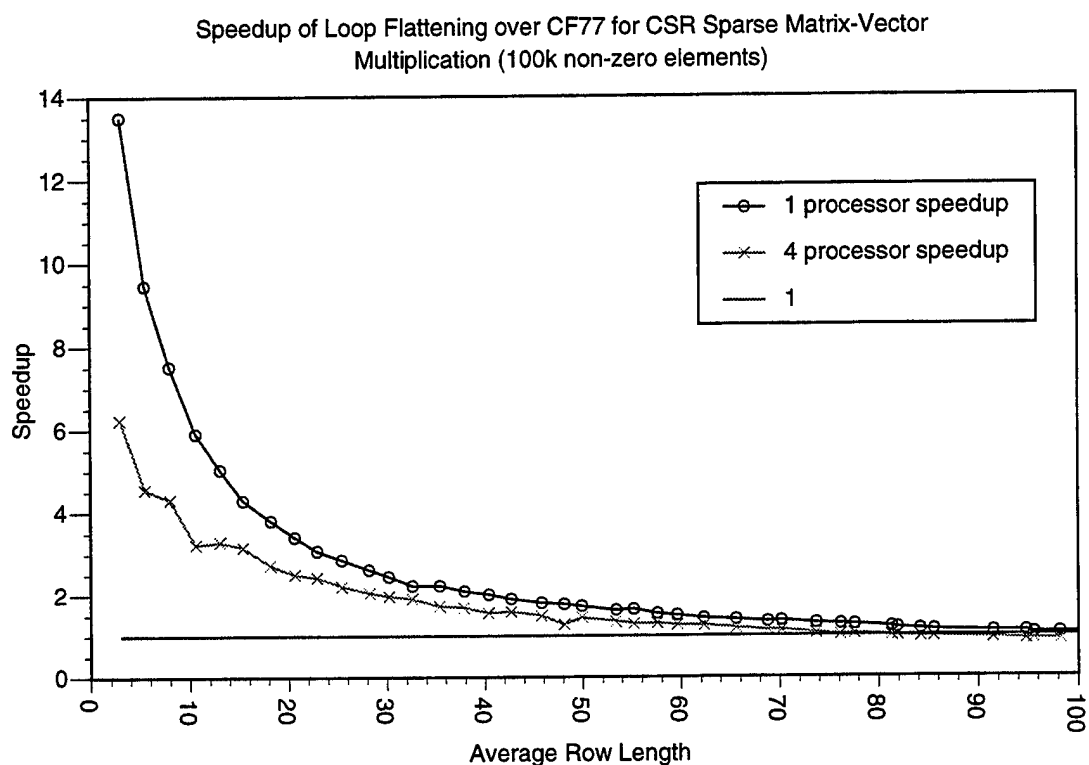
Figure 8.15 Relative speedup of CSR sparse matrix-vector multiplication kernel.

Speedup of Loop Flattening over CF77 for CSC Sparse Matrix-Vector
Multiplication (100k non-zero elements)

Figure 8.16 Relative speedup of CSC sparse matrix-vector multiplication kernel.

137

The peak per head performance our compiler generates for CSR now stands at around 75 MFLOPS for this application on multiple processors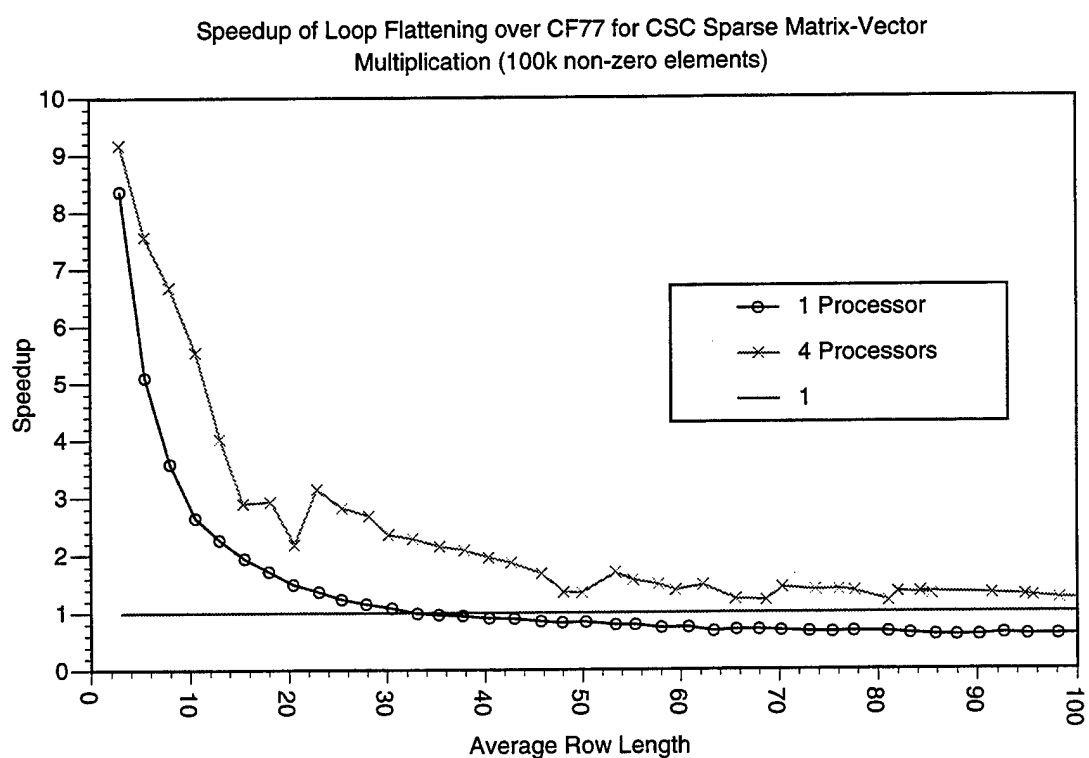, with a typical sustained rate of a little over 117 MFLOPS on a single processor. The best current single head performance for this particular C90 configuration is a little over 170 Mflops using the assembly level SEGMULV routine. So, despite the obvious improvement over current automatic parallelization techniques, there is significant room for improvement in the back end of the compiler. We believe that some performance improvement can be realized by directly generating assembly code for portions of the reduction, scan, and combining-send code templates. For example, the Fortran version of the optimized library is around 25% slower than the assembly version. Furthermore, at the source level, there are still some code template optimizations which can be employed (i.e. better operator selection). As mentioned earlier, there is some intrinsic overhead in using the recurrence parallelization technique in compiler that can likely be made up by optimizing the templates.

For the CSC kernel, the speedup for 100,000 elements is plotted in figure 8.16. The performance on a single and four vector processors is plotted in figure 8.13 and 8.14. The crossover point for a single processor is at an average row length of 10-11, with a peak relative speedup of 3. The peak relative speedup on 4 processors is 9, with no crossover point. The typical sustained rate per processor on multiple processors is nearly 30 MFLOPS, while for a single processor the peak is 50 MFLOPS. The CSR kernel performance is more than twice as fast. When explicitly parallelizing a CSC kernel, a good approach might be to transform representations so that a CSR library routine may be used. The reason for the failure to do this in an automated fashion is **not** intrinsic to the compilation technique. It can be overcome easily in most cases. The problem is that we do not subject the templates for reduction, scan, combining-send and multiprefix to compiler optimization.

We currently only attempt to move the loop flattening overhead out of surrounding loops. However, we can also move portions of the particular combining-send operator out of surrounding loops, so that the cost may be amortized over many matrix multiplications. For the multiprefix algorithm that we currently use, a good deal of time of the algorithm is spent processing the index array to construct the SPINETREE structure on which we perform what are essentially multiple simultaneous reductions and scans [76]. This phase can be amortized over multiple invocations of the kernel by hoisting it out of surrounding loops. For a sorting version of a combining-send or multiprefix, the sort can be can be performed once for multiple invoca-

138

tions of the parallelized loop nest. The performance of the CSC kernel should more closely approximate that of the compiled CSR kernel using this approach. This is especially true for the sorting approach (all that would be left after this optimization would be a reduction or scan), which would otherwise be slower than the approach we are currently using. Thus, some flexibility in code template selection may be necessary.

The overhead for the index set computation preamble is equivalent to about a single sparse matrix-vector multiplication, and in both cases is hoisted out of surrounding loops.

In the code template for reductions and scans, there is an inherently serial sum across a vector register length for each processor. Currently, this entire portion of the computation is serialized. However, the sum can be performed on each processor in parallel, and then combined serially across the processors. For this particular portion of the computation, the speedup should approach the vector register length of the machine. The combining-send/multiprefix code template does not task as well as the scan and reduction templates, because of indirection and added synchronization overhead.

8.4.4 Bucketsort

The NAS benchmark suite includes a bucketsort routine to perform integer sort. This is essentially a single pass radix sort. The code we are concerned with does not actually perform the permute for the sort, it only performs the ranking step. The code for this routine:

```
      subroutine bucksort(key, rank, keyden, N, MAXKEY)
      integer          N
      integer          MAXKEY
      integer          key(0:N)
      integer          rank(0:N)
      integer          keyden(0:MAXKEY)
      integer          i, j, k

      do 40 i=1, MAXKEY
           keyden(i) = 0
 40        continue

      do 60 i=1, N
           k = key(i)
           keyden(k) = keyden(k) + 1
 60        continue

      do 80 i=2, MAXKEY
```
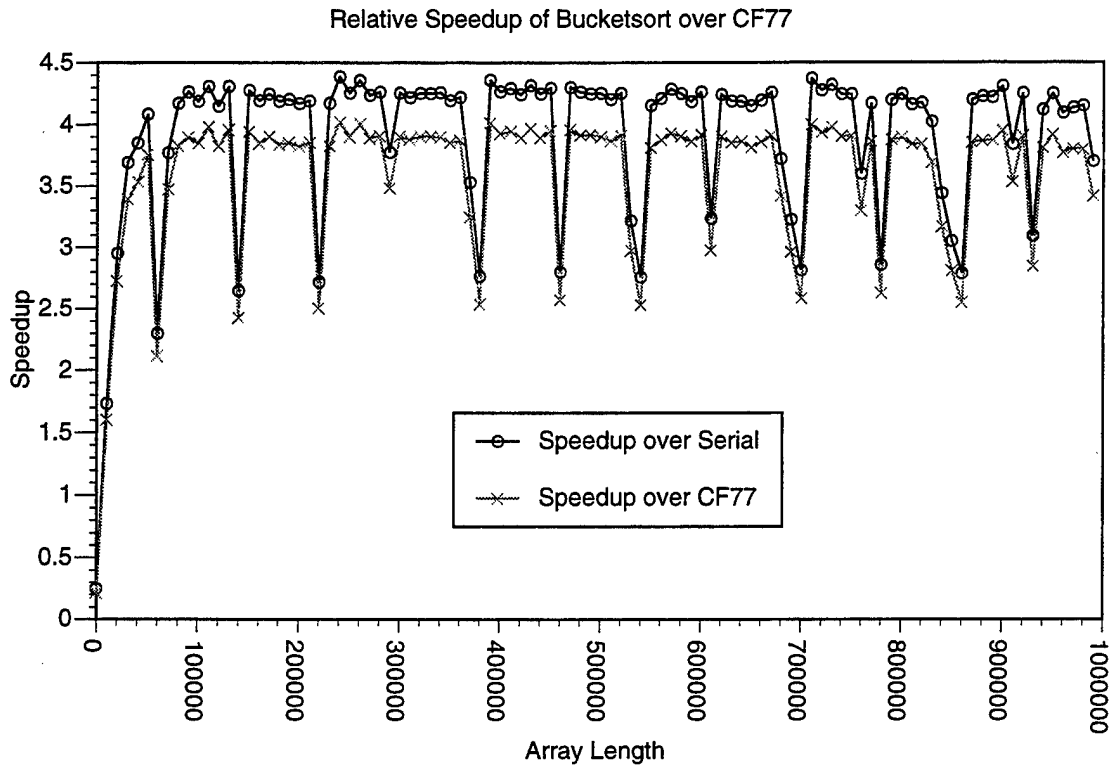
Relative Speedup of Bucketsort over CF77

Figure 8.17 Ranking performance of bucketsort from NAS benchmark suite.

```
                keyden(i)  = keyden(i)  + keyden(i-1)
80        continue

      do  110 i=1, N
            k = key(i)
            keyden(k)  = keyden(k)  - 1
            rank(i)    = keyden(k)
110       continue
      return
      end
```

The first loop is trivially parallelizable. The second loop is parallelizable by a simple histo-gramming combining-send. The third loop is an inclusive sum scan. The fourth loop is essen-tially a simple multiprefix operation. The CF77 compiler can only parallelize the first and second loops. Our compiler parallelizes all the loops in the routine. The absolute and relative performance for this program is plotted in figures 8.17 and 8.18. The speedup over CF77 ranges between 5 and 7 for reasonably large key counts. Over a range of key densities for the sorted keys, the performance is charted in figure 8.19.

Relative Speedup of Bucketsort over CF77
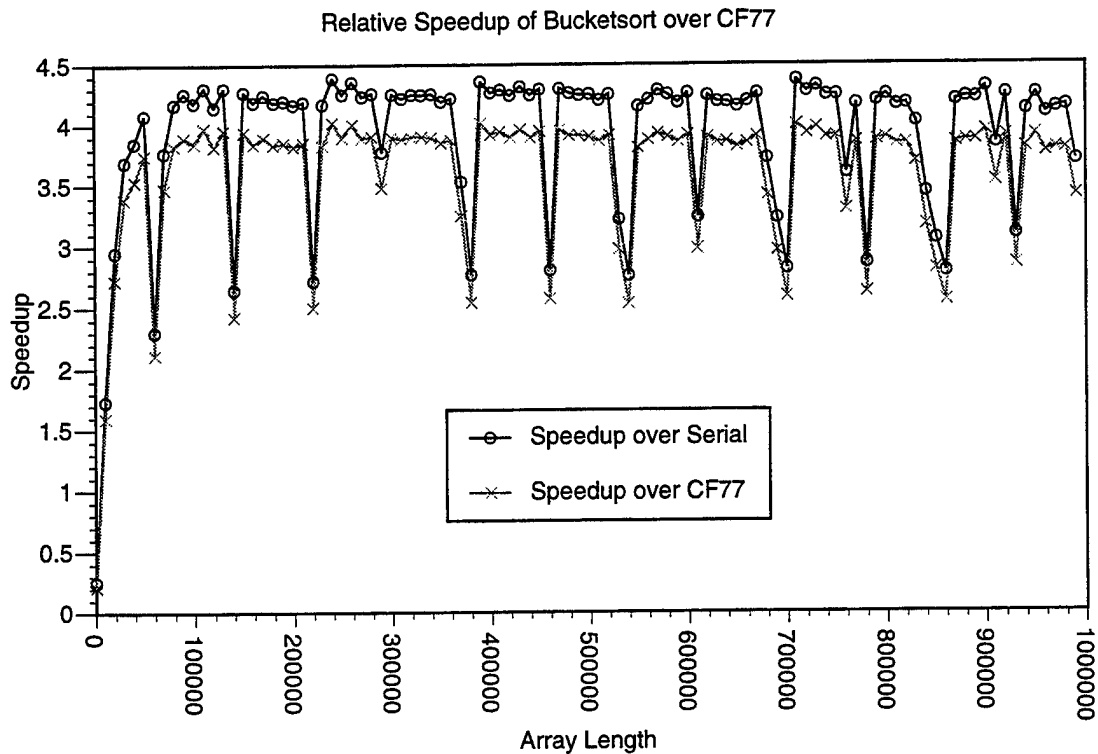

Figure 8.18 Relative speedup of bucketsort from NAS benchmark suite.

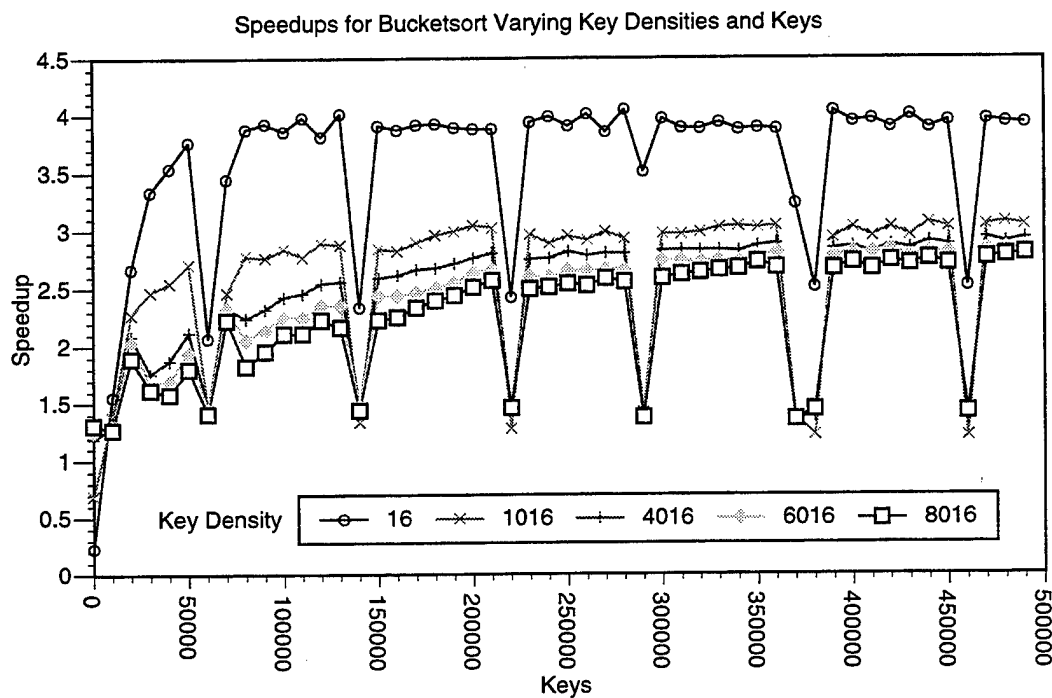Speedups for Bucketsort Varying Key Densities and Keys

Figure 8.19 Relative speedup of bucketsort from NAS benchmark suite for varying key densities.
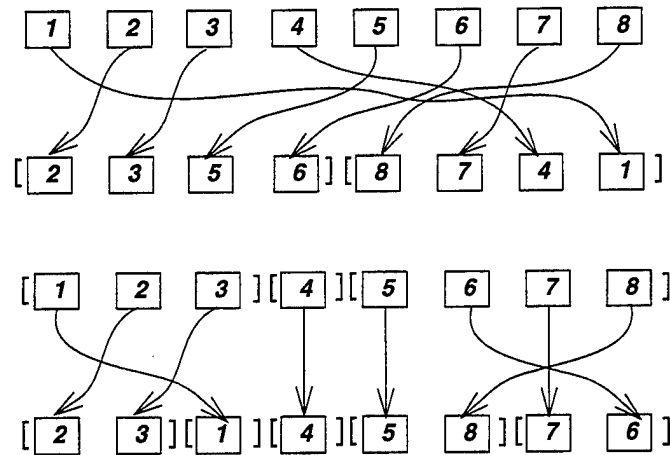
Figure 8.20 Partitioning patterns for simple (top) and segmented (bottom) partition operation.

8.4.5 Partition

Partition is a pair of simple loops which divides the elements of an array into different partitions based on their values. The general pattern of the computation is illustrated in figure 8.20. This type of code is prevalent in divide-and-conquer style programs. The first loop works on arrays with single partitions:

```
pivot = a(1)
lower = 1
upper = size
do i = 1, size
   if (a(i) .lt. pivot) then
      b(lower) = a(i)
      lower = lower + 1
   else
      b(upper) = a(i)
      upper = upper - 1
   endif
enddo
```

The relative and absolute performance of this loop as parallelized by our compiler is plotted in figures 8.21 and 8.22. CF77 is unable to parallelize this loop. Our compiler parallelizes this loop by precomputing the monotonic induction variables `upper` and `lower` using scan operations. Those arrays are used to permute the values of `a` into the array `b`. Note, however, the detail of the relative performance of this parallelized loop for small sequences plotted in figure 8.23. This will play an important role when we later consider trade-offs in embedding control in divide-and-conquer algorithms.
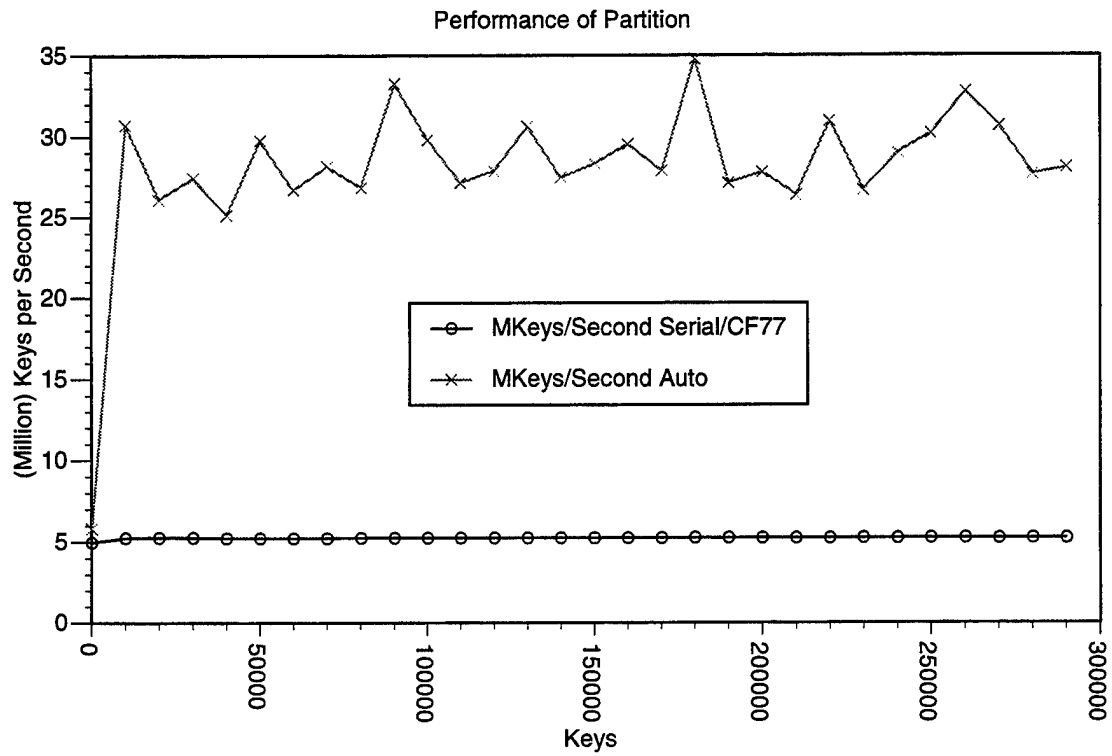
142

Performance of Partition

Figure 8.21 Performance of simple partition loop.

Speedup over Serial/CF77 of Partition
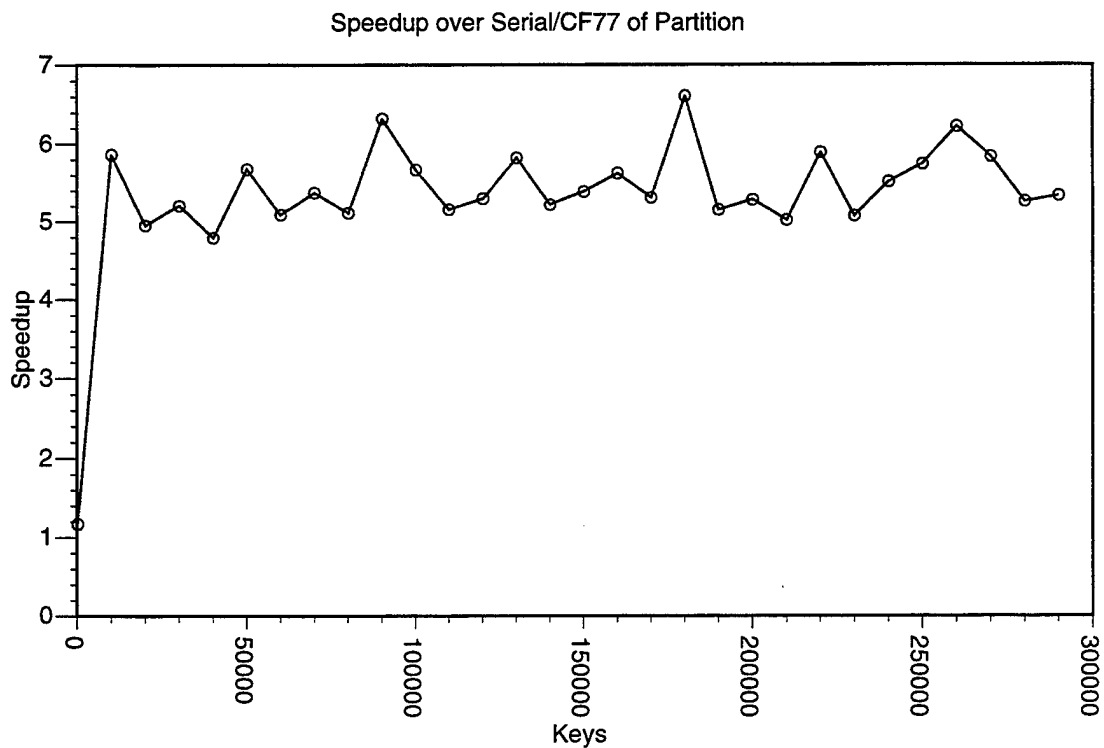
Figure 8.22 Relative speedup of simple partition loop.

Performance Degradation Relative to CF77 in Simple Partition at Low Sequence Lengths
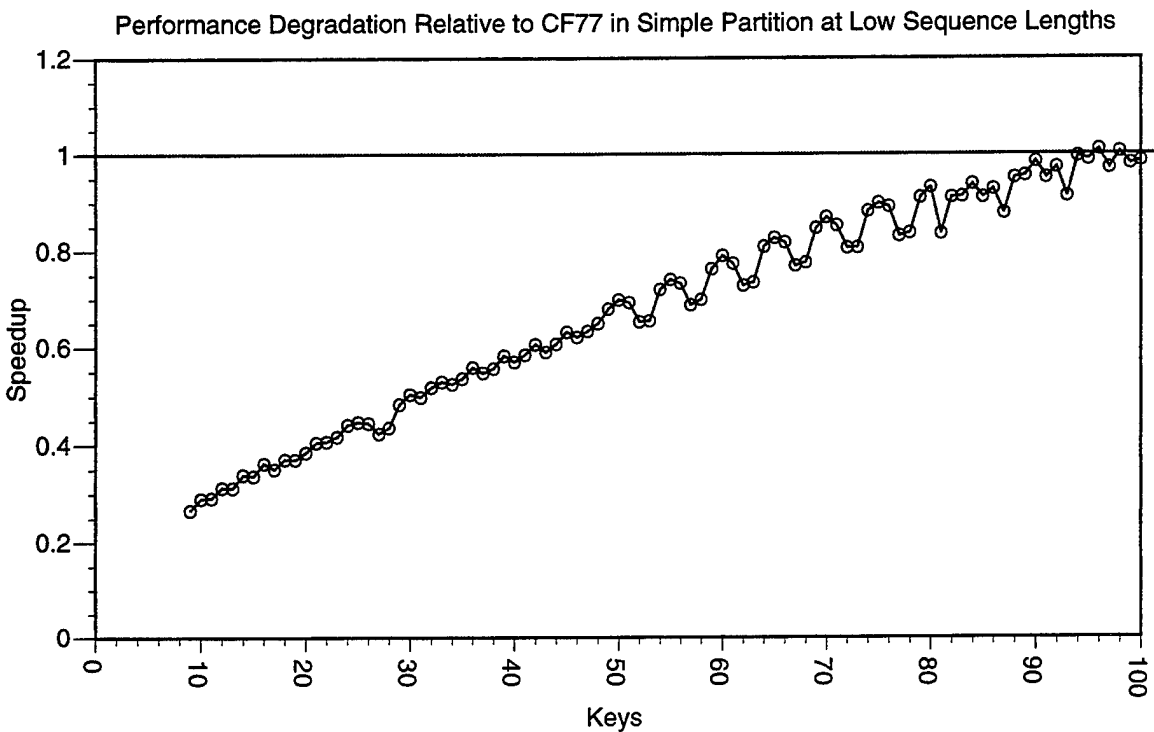
Figure 8.23 Relative speedup (slowdown) of partition on short sequences.

The second loop works on arrays with an arbitrary number of partitions. This is referred to as a segmented partition:

```
do fxp = 1, fxpartitions
    pivot = a(begin(fxp))
    lower = begin(fxp)
    upper = end(fxp)
    do i = begin(fxp),end(fxp)
        if (a(i) .lt. pivot) then
            b(lower) = a(i)
            lower = lower + 1
        else
            b(upper) = a(i)
            upper = upper - 1
        endif
    enddo
enddo
```

Begin and end are instantiated with the beginning and ending index of each partition. This loop has irregular structure and is parallelized with what is effectively a segmented scan via loop flattening. The relative and absolute performance of this loop as parallelized by our compiler is plotted in figures 8.24 and 8.25. The speedup over CF77 drops from 5-7 in the non-

144

Performance of Segmented Partition of 200000 Key Sequence over varying partition sizes

Figure 8.24 Performance of segmented partition loop over a range of segmentation factors.

Speedup over CF77 on Segmented Partition over Varying Numbers of Partitions
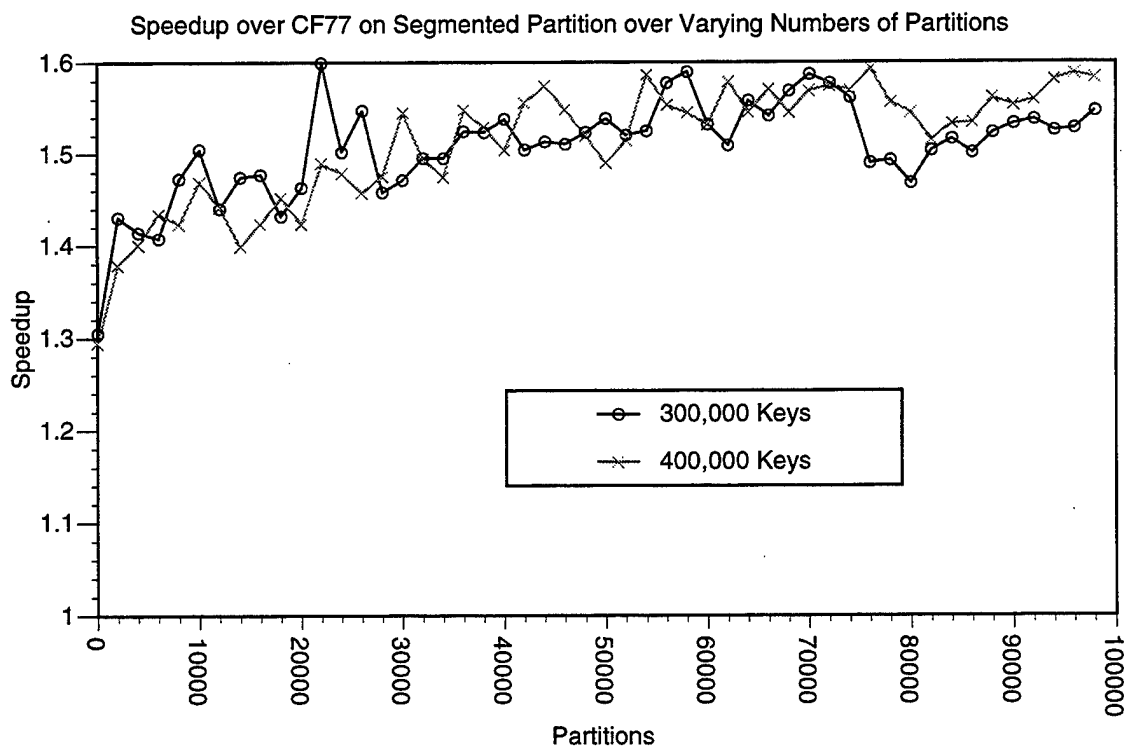
Figure 8.25 Relative speedup of segmented partition loop over a range of segmentation factors.

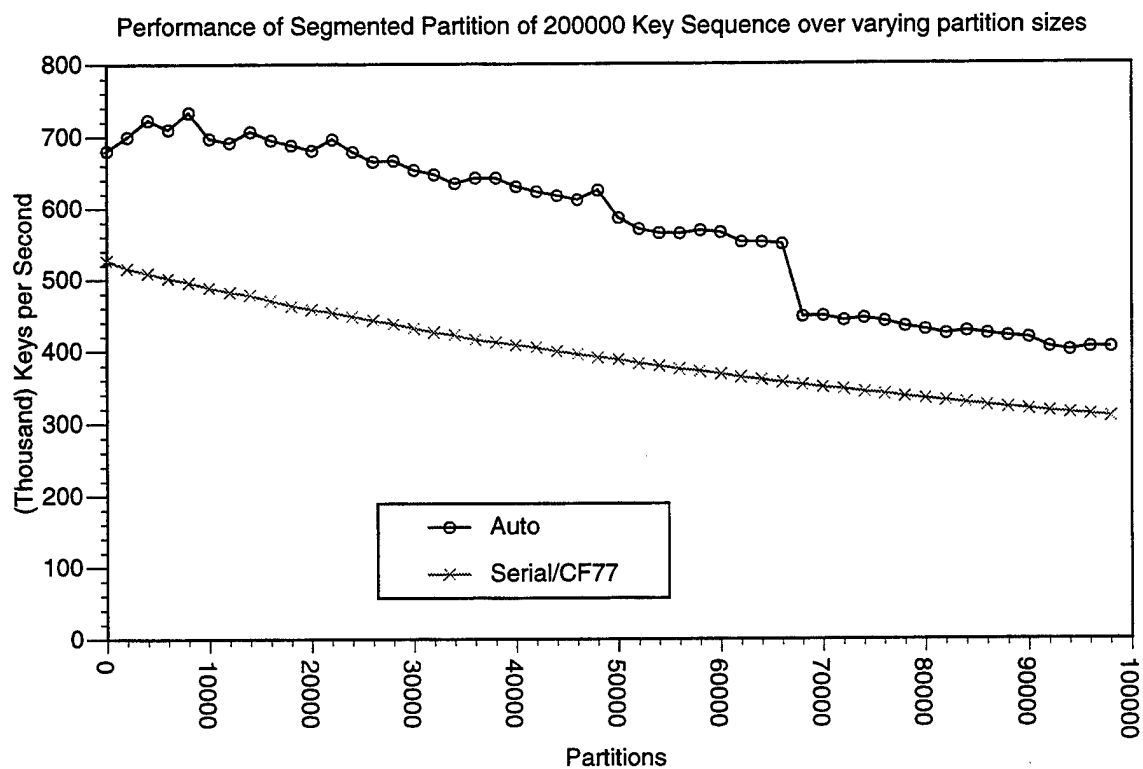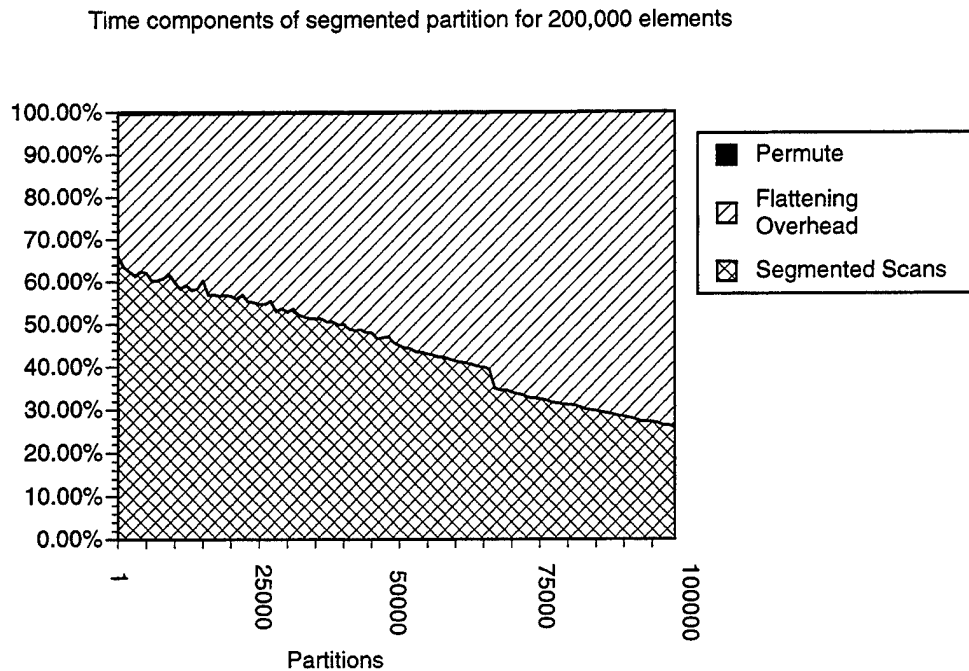Time components of segmented partition for 200,000 elements

Figure 8.26 The contribution of segmented scans and flattening overhead to the execution time of segmented partitions.

segmented case to 1.3-1.5 in the segmented case. The overhead of loop flattening is included in our measurements since it is unlikely that this overhead can be amortized over multiple instances of this loop. The typical setting for this code is recursive subroutine which has had control embedded. There is no opportunity to hoist out the loop flattening overhead in any invocations of the subroutines.

8.4.6 Flattening Overhead in Partition Loops

There are several causes of the degradation in performance from the simple partition to the segmented partition loops. A breakdown of the components which contribute to the execution time of the segmented partition are shown in figure 8.26. In segmented partition, as is evident from this figure, and simple partitions the data movement (permute) step accounts for a negligible amount of time. The loop flattening overhead, which is not paid in the case of a simple partition contributes between 30 and 66 percent of the execution time. The segmented scan operations in the segmented partition account for the balance of the time. In the case of a simple partition, scans dominate the execution time since there is no flattening overhead. The loop flattening overhead of computing loop indices and the segmented scans vary in their relative contribution to execution time by the fragmentation of the segmented operation. As there are more partitions introduced into the sequence, the cost of the loop flattening overhead

Overhead of Segmentation in a Simple Scan
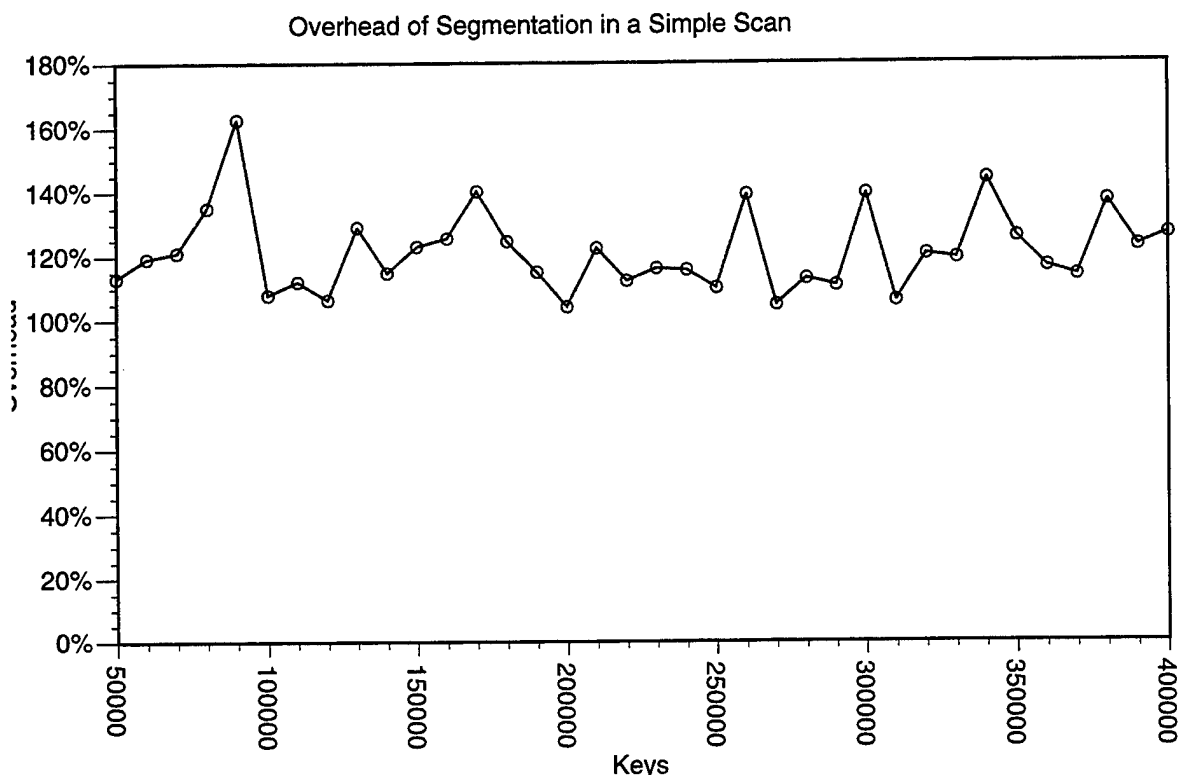
Figure 8.27 The overhead of segmentation in a simple scan from the partition loops.

increases with respect to the segmented scan overhead. The amount of work in the segmented scans decreases in the segmented partition loop because the number of 1 and 2 element partitions becomes a significant factor in the overall effective sequence length the flattened segmented operation works on. Furthermore, there is a slight increase in execution time of the flattening overhead template as the segmentation of the inner loop trip counts become smaller.

The relative overhead of segmented versus non-segmented operations is plotted in figure 8.27. The segmented operations are consistently about a factor of 2 slower than their non-segmented counterparts. The primary reason is the added computation due to the inclusion of segment information (derived implicitly in the composition operator for the recurrent loop). This overhead can be reduced by using more efficient representation and computation schemes for segmentation, rather than costly arithmetic operations. For example, explicit conditionals can be used in bodies of segmented operations instead of the integer addition and multiplication which is introduced for each element of the sequence by our analysis. The fortran version of the hand-written sparse matrix-vector multiplication routine in section 8.4.3 uses such a

Figure 8.28 Partitioning patterns through several steps of a hypothetical divide-and-conquer algorithm.
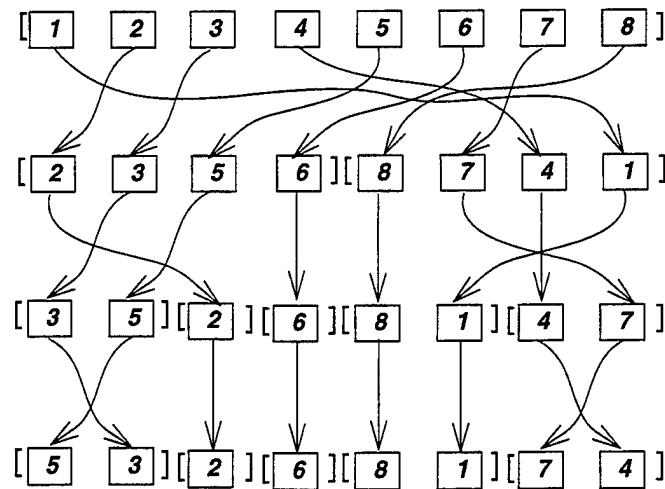
scheme, as well as more efficient packed bit-vector representation for segment descriptors. This would be easy to introduce as a post-parallelization code optimization.

The relative contributions of loop flattening overhead can be mitigated in a number of ways. Reuse is the most important mechanism by which this overhead can be effectively reduced. In the sparse matrix-vector multiplication example of section 8.4.3, the overhead was reusable because the kernel was embedded within a loop which multiplied many vectors. The compiler was able to hoist this flattening overhead out of surrounding loop control. Furthermore, in cases where multiple loops are flattened within a procedural context, some loop flattening overhead may be reused. This plays an important role in the divide-and-conquer algorithms we discuss next. Finally, there are opportunities for interprocedural reuse of loop flattening overhead, especially in divide-and-conquer style recursion. We will discuss this further at the end of this chapter.

8.4.7 Simple Quicksort

Simple quicksort is a non-stable sort which works only for sequences of nonrepeating keys. Though it has limited application, its simplicity will be contrasted with that of the more complex quicksort: The routine divides the partition into two parts based on a simple pivot selection. Then the partitioned array is copied back into the source array and each partition is recursed upon. The progression of the algorithm is illustrated in figure 8.28.

```
recursive subroutine qsort_serial(a,b,begin,end,n)
integer begin,end,n
```

```
      integer a(n),b(n)
      integer lower, upper, i

      if ((end - begin) .le. 1) then
         return
      endif

      pivot = a(begin)
      lower = begin
      upper = end
      do i = begin,end
         if (a(i) .lt. pivot) then
            b(lower) = a(i)
            lower = lower + 1
         else
            b(upper) = a(i)
            upper = upper - 1
         endif
      enddo

      do i = begin, end
         a(i) = b(i)
      enddo

      call qsort_serial(a,b,begin,lower-1,n)
      call qsort_serial(a,b,upper+1,end,n)
      end
```

When control is embedded the partition loop here is transformed to a segmented partition is parallelized as in section 8.4.5. Figures 8.29 and 8.30 plot the relative and absolute performance of this quicksort using our compiler. The sustained speedup over CF77 hovers about 1.6.

Control embedding creates segmented partition operations in these divide and conquer algorithms. Without control embedding, our compiler would simply parallelize the recursive subroutine's body, parallelizing a simple partition. Turning off control embedding and simply parallelizing the subroutine body gives the relative performance speedup plotted in figure 8.31. The compiler in this case has fully parallelized the subroutine body. The reason for this degradation in performance is that the parallelism available within each partition quickly gets very small. The detail on the parallelized partition loop performance is plotted in figure 8.32. The poor performance at small sequences is the primary factor in the poor relative performance the parallelized code without control embedding. It is also important to note that parallelizing without control embedding does not scale to multiple vector processors as well as the

Quicksort - Nonrepeating Keys - Keys per Second
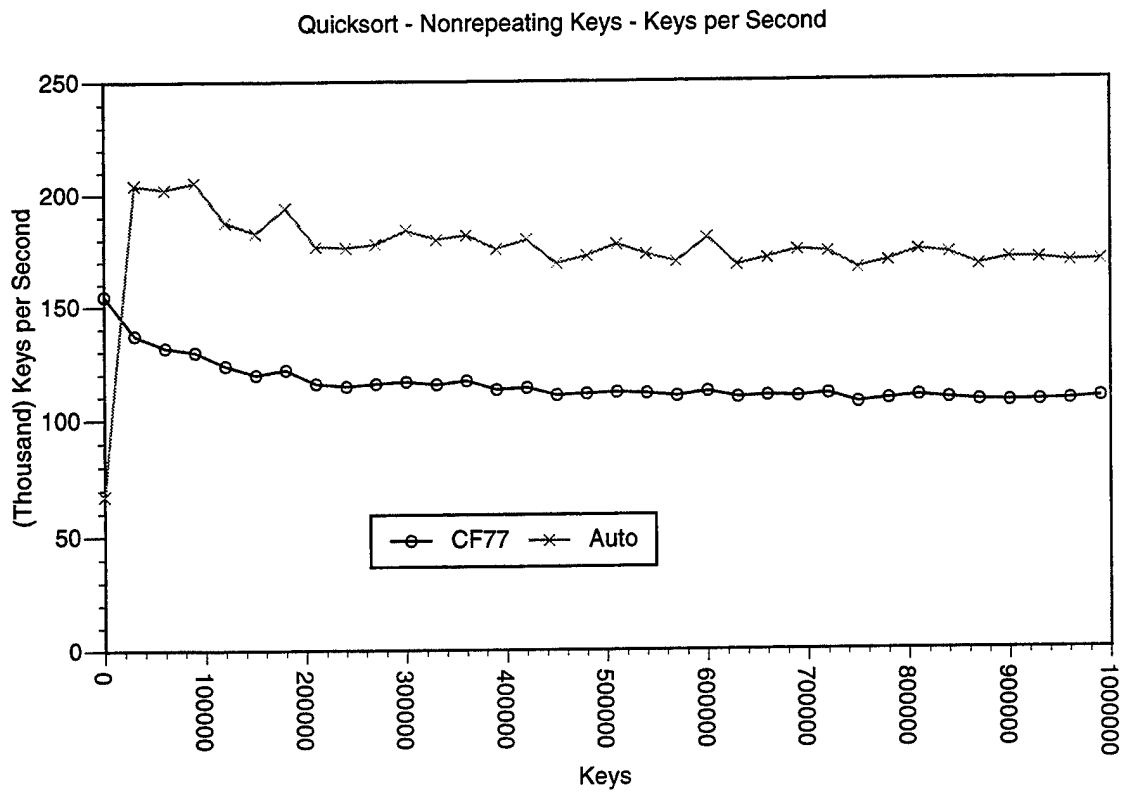
Figure 8.29 Performance of simple quicksort.

Quicksort (Nonrepeating Keys) - Speedup
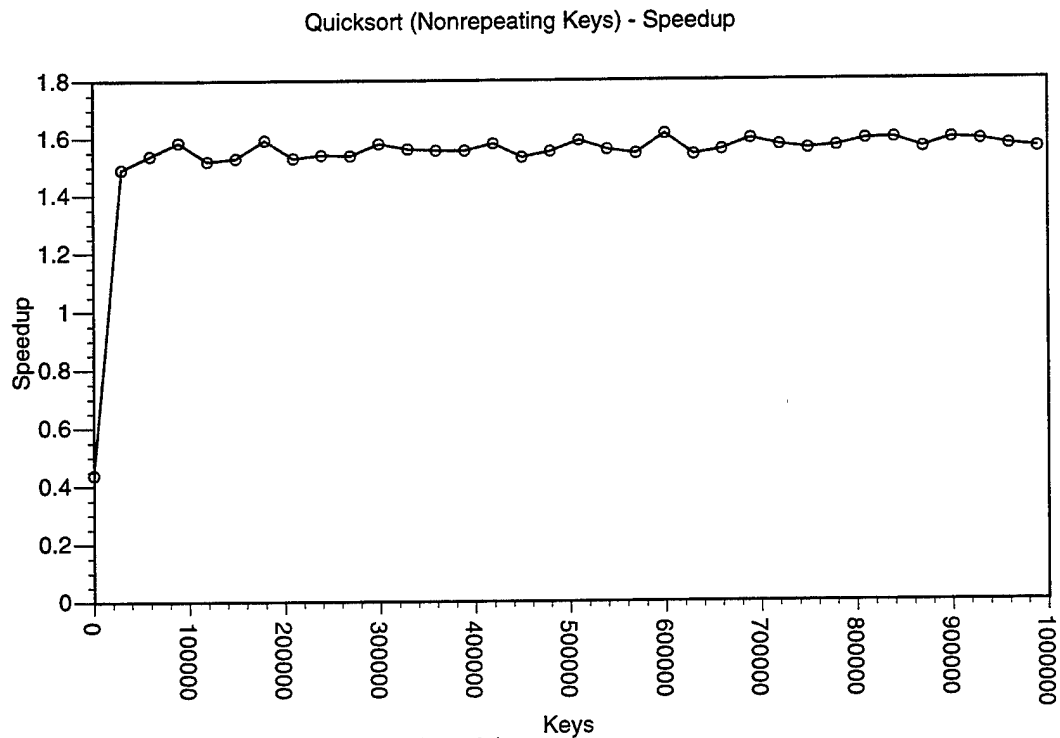
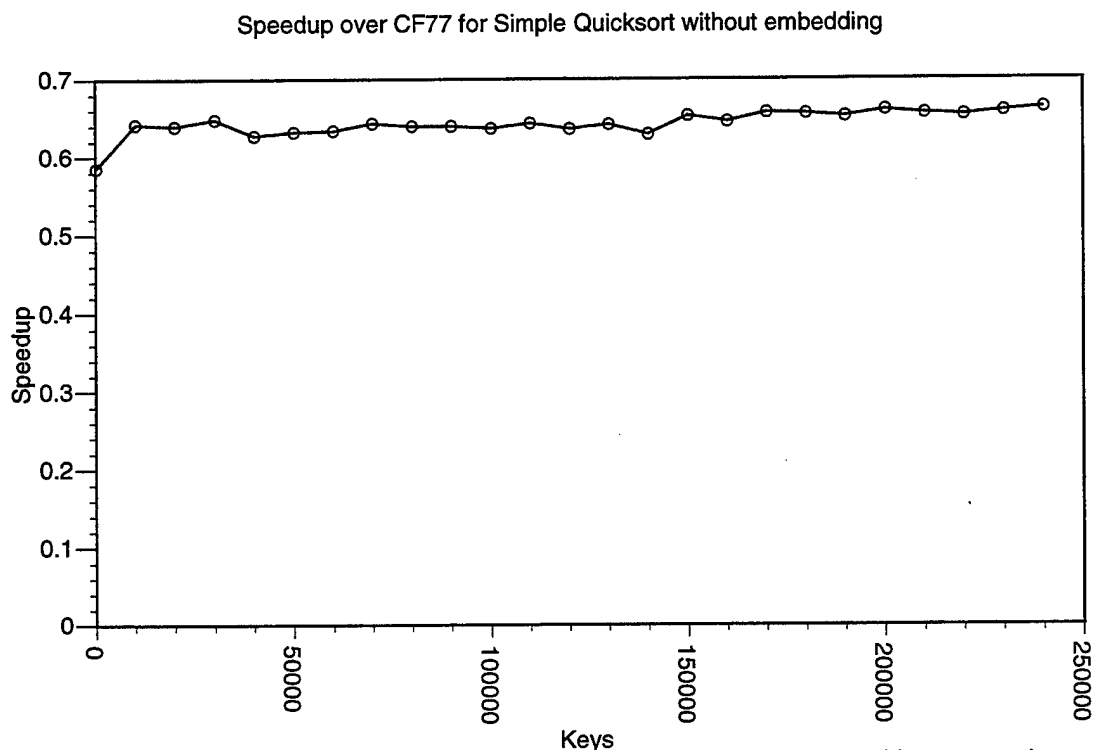Figure 8.30 Relative speedup of simple quicksort.

150

Speedup over CF77 for Simple Quicksort without embedding

Figure 8.31 Performance degradation of fully parallelized simple quicksort without control embedding.

Performance Degradation Relative to CF77 in Simple Partition at Low Sequence Lengths
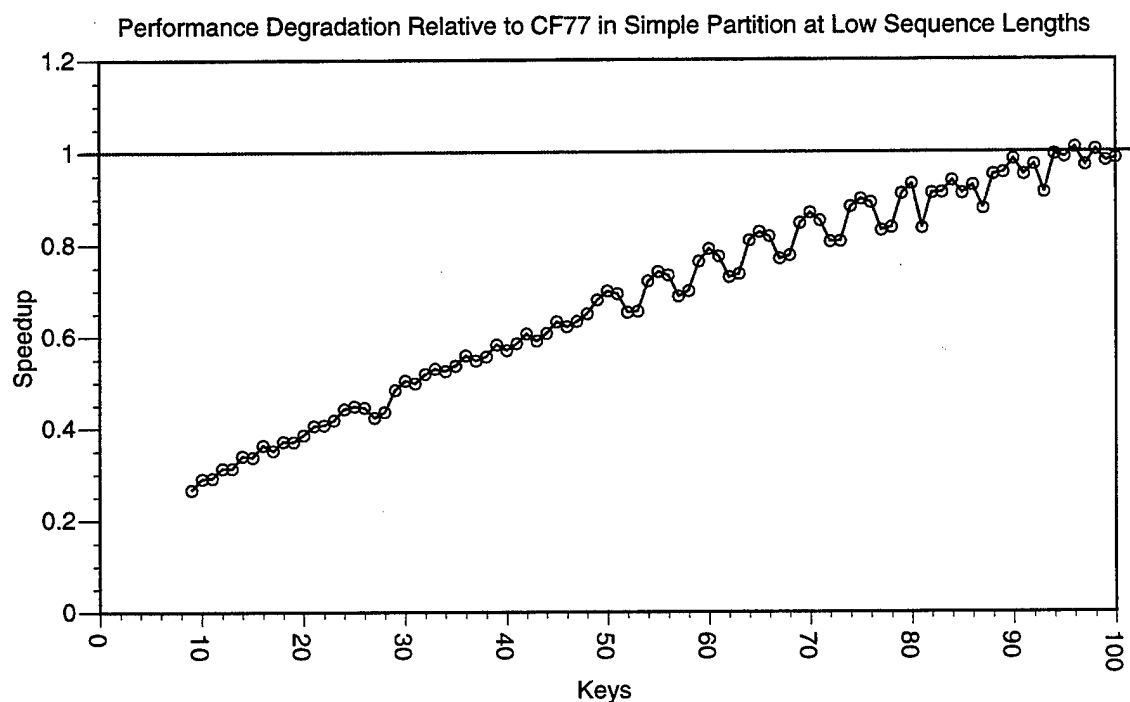
Figure 8.32 Detail on the relative performance of the parallelized partition loop for small sequence lengths.

Sorting Rate for Simple Quicksort without Control Embedding
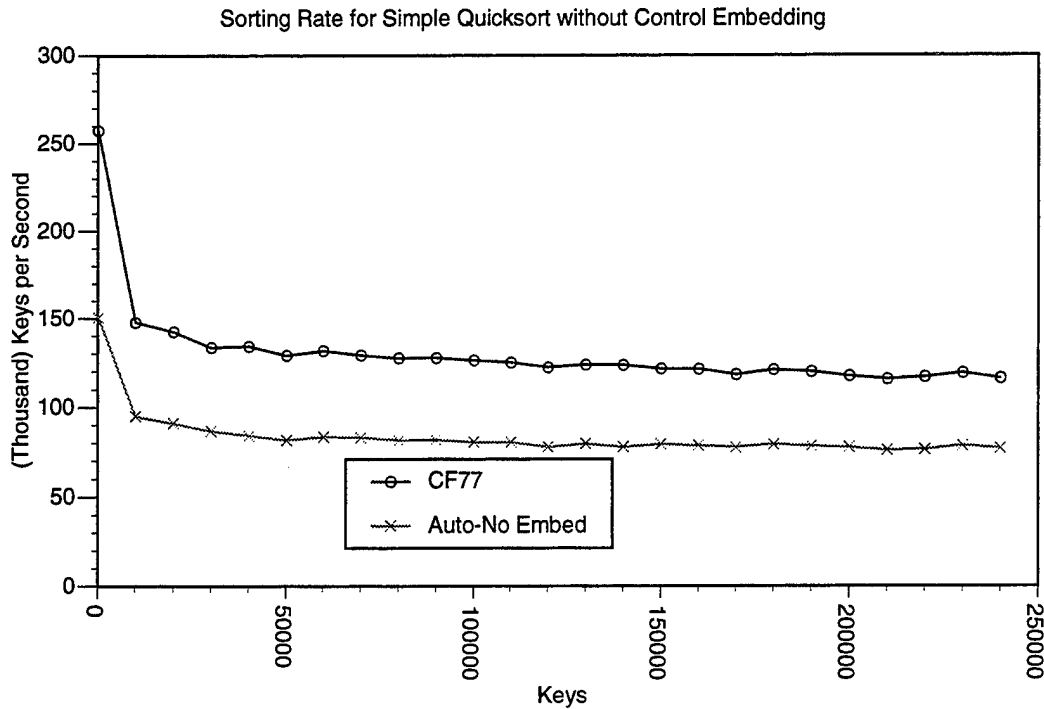
Figure 8.33 Performance of simple quicksort without control embedding with CF77-generated code performance.

control embedding approach. Furthermore, the absolute performance degrades significantly without control embedding, plotted in figure 8.33. To improve upon this, we might mix the serial and parallelized code, resorting to serial code as the partitions grow smaller, as in figure 8.34. The performance still lags behind the CF77 code, primarily due to the added cost of the conditional to check whether the size is large enough to invoke the parallel partition. This approach will not scale very well either, though the performance on a single vector processor clearly improves.

8.4.8 Stable Quicksort

The general, stable version of quicksort partitions by the pivot into three portions, those elements less than, equal to, and greater than the pivot. This requires two loops. The first loop counts the elements in the first two partitions to set up the proper values for the monotonic induction variables lower, middle, and upper. The partitioning loop is a straightforward adaptation of the partition loop of the simple quicksort. Only the first and last partitions are recursed upon.

```
recursive subroutine qsort(a,b,begin,end,n)
integer begin,end,n
integer a(n), b(n)
```
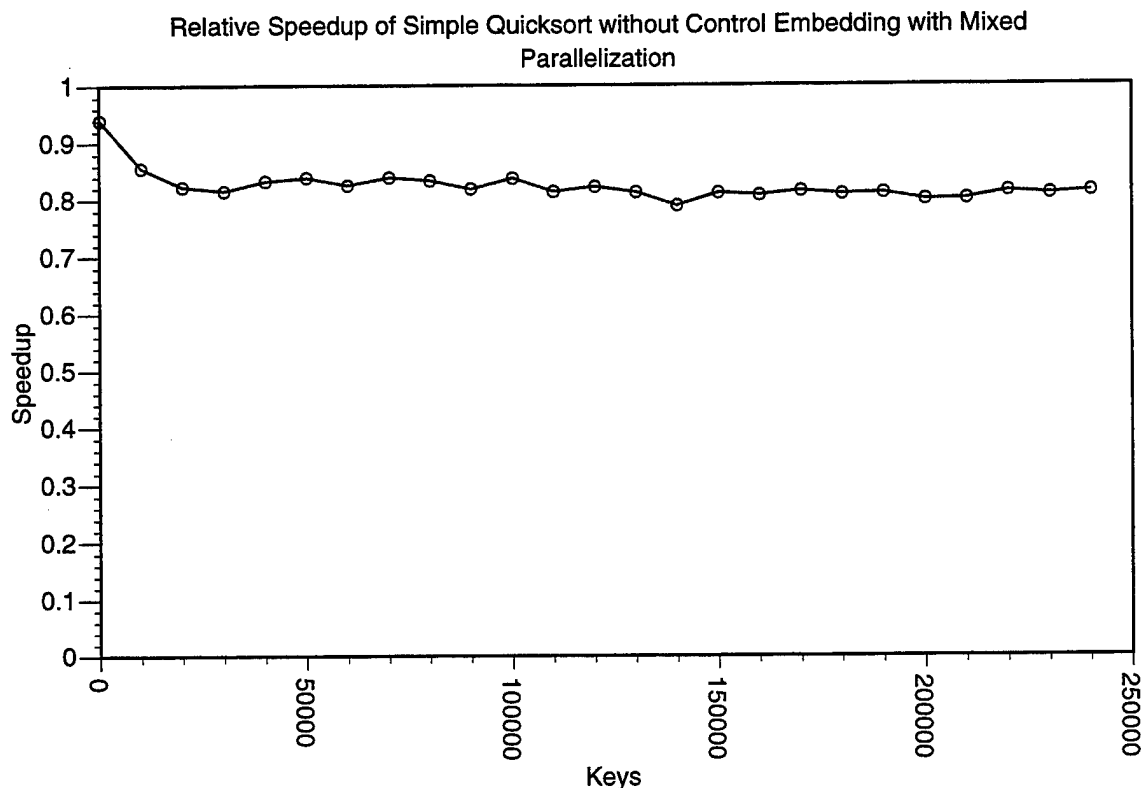
Figure 8.34 Speedup of simple quicksort without control embedding with mixed parallelization and serialization relative to CF77-generated code performance.

```fortran
integer lower, upper, middle, i

if ((end - begin) .le. 1) then
   return
endif
pivot = a(begin)
upper = begin
middle = begin
lower = begin
do i = begin, end
   if (a(i) .lt. pivot) then
      middle = middle + 1
      upper = upper + 1
   else if (a(i) .eq. pivot) then
      upper = upper + 1
   endif
enddo
do i = begin,end
   if (a(i) .lt. pivot) then
      b(lower) = a(i)
      lower = lower + 1
   elseif (a(i) .gt. pivot) then
      b(upper) = a(i)
```

Quicksort - Stable - Keys per second
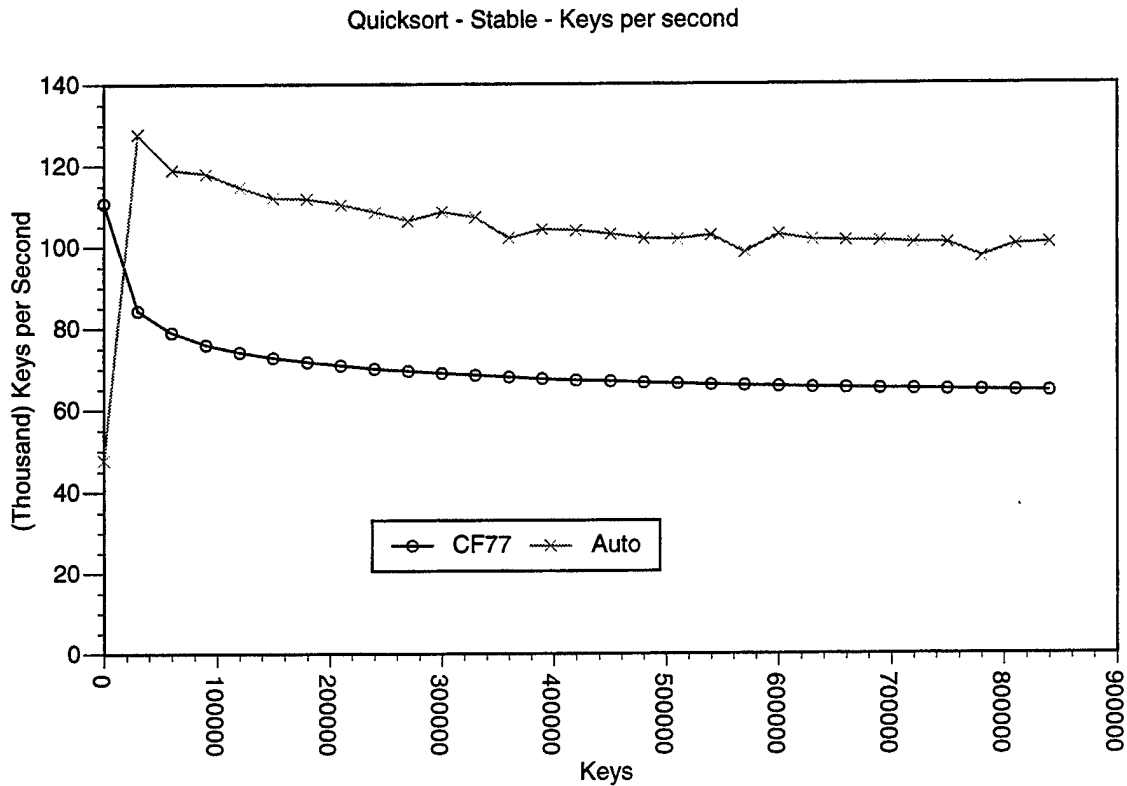
Figure 8.35 Performance of stable quicksort.

```
      upper = upper + 1
   else
      b(middle) = a(i)
      middle = middle + 1
   endif
enddo
do i = begin,end
   a(i) = b(i)
enddo

call qsort(a,b,begin,lower-1,n)
call qsort(a,b,middle,end,n)
end
```

The first loop computes offsets for each partition's monotonic induction variable and is very similar to the partition loop, except that it does not require a permutation and the primitive used is a reduction (segmented). Since the index computations are identical, the loop flattening overhead of this loop can be reused for the two subsequent loops. This keeps the relative performance improvement over CF77, plotted in figure 8.36 nearly identical to that of the simple quicksort. The absolute performance is plotted in figure 8.35.

Quicksort - Stable - Speedup
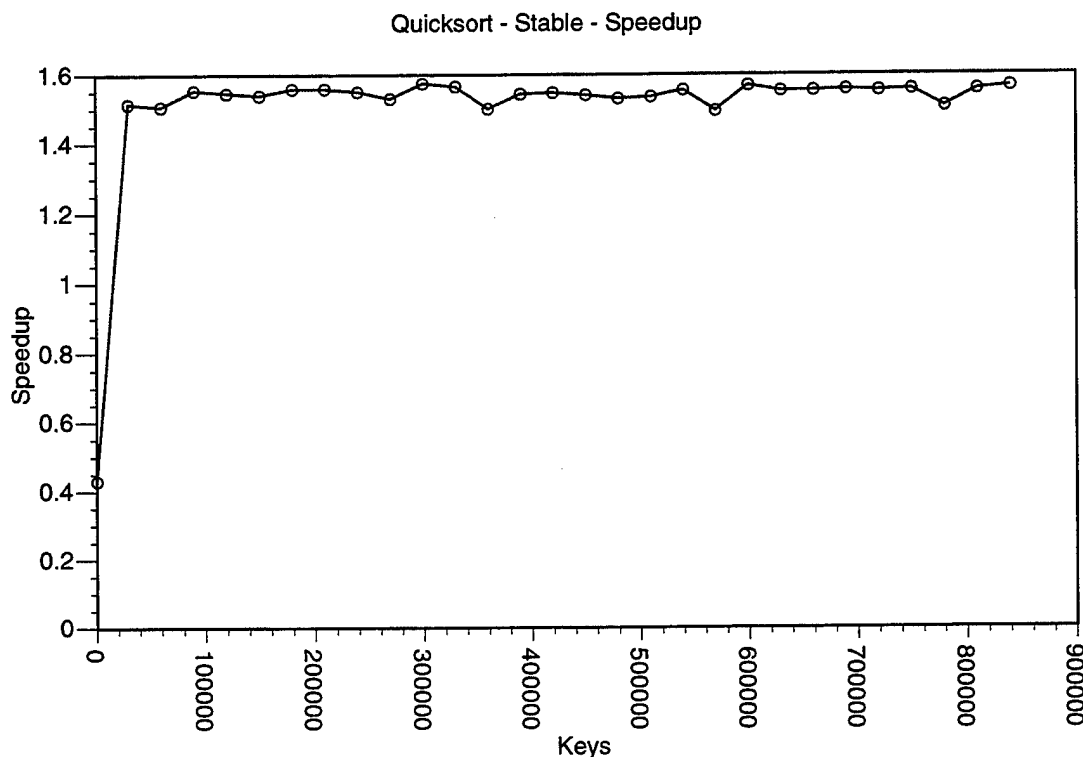
Figure 8.36 Relative speedup of stable quicksort.
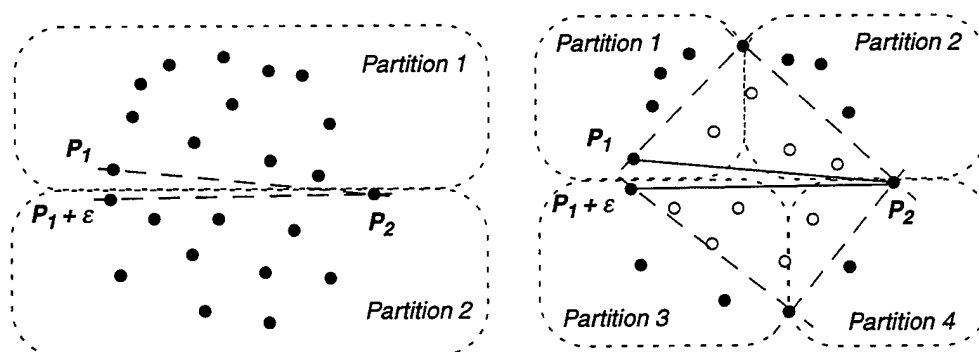
Figure 8.37 Partitioning in the quickhull algorithm, with filled dots denoting points which are still under consideration for inclusion in convex hull.

8.4.9 Planar Quickhull

The final divide-and-conquer algorithm we consider is a quickhull routine [68]. While most partition steps in divide-and-conquer algorithms will likely resemble those in the quicksort examples, the planar quickhull partition step is slightly different. Rather than a partition of all the points, the algorithm eliminates points it concludes are definitely not on the convex hull and divides the remaining points. This partitioning is illustrated in figure 8.37. More importantly, the algorithm reveals more loop flattening possibilities, exposing the pitfall of this con-

155

trol structure transformation approach. Another difference is the non-trivial merge step after the recursive calls, which merges the partial convex hull chains computed in each call. The code for this algorithm is below:

```
      recursive subroutine quickhull_serial(xa,ya,xb,yb,begin,
$ end,count,n)
      integer begin,end,count,n
      integer xa(n),ya(n),xb(n),yb(n)
      integer maxp,maxcross,count1,count2,cross,p

      if (end-begin.le.1) then
         count = end - begin + 1
         return
      endif
      p = begin
      maxp = begin
      maxcross = 0
      do i = begin, end
         cross = (xa(begin)-xa(i))*(ya(end)-ya(i)) -
$            (ya(begin)-ya(i))*(xa(end)-xa(i))
         if (cross.ge.0) then
            xb(p) = xa(i)
            yb(p) = ya(i)
            p = p + 1
         endif
         if (maxcross.lt.cross) then
            maxcross = cross
            maxp = p - 1
         endif
      enddo
      if (maxp.eq.begin) then
         maxp = maxp + 1
      else if (maxp.eq.end) then
         maxp = maxp - 1
      endif
      call quickhull_serial(xb,yb,xa,ya,begin,maxp,count1,n)
      call quickhull_serial(xb,yb,xa,ya,maxp,p-1,count2,n)
      do i = 1,count1
         xa(begin+i-1) = xb(begin+i-1)
      enddo
      do i = 1,count2-1
         xa(begin+count1+i-1) = xb(maxp+i)
      enddo
      count = count1 + count2 - 1
      end
```

The Performance of Quickhull
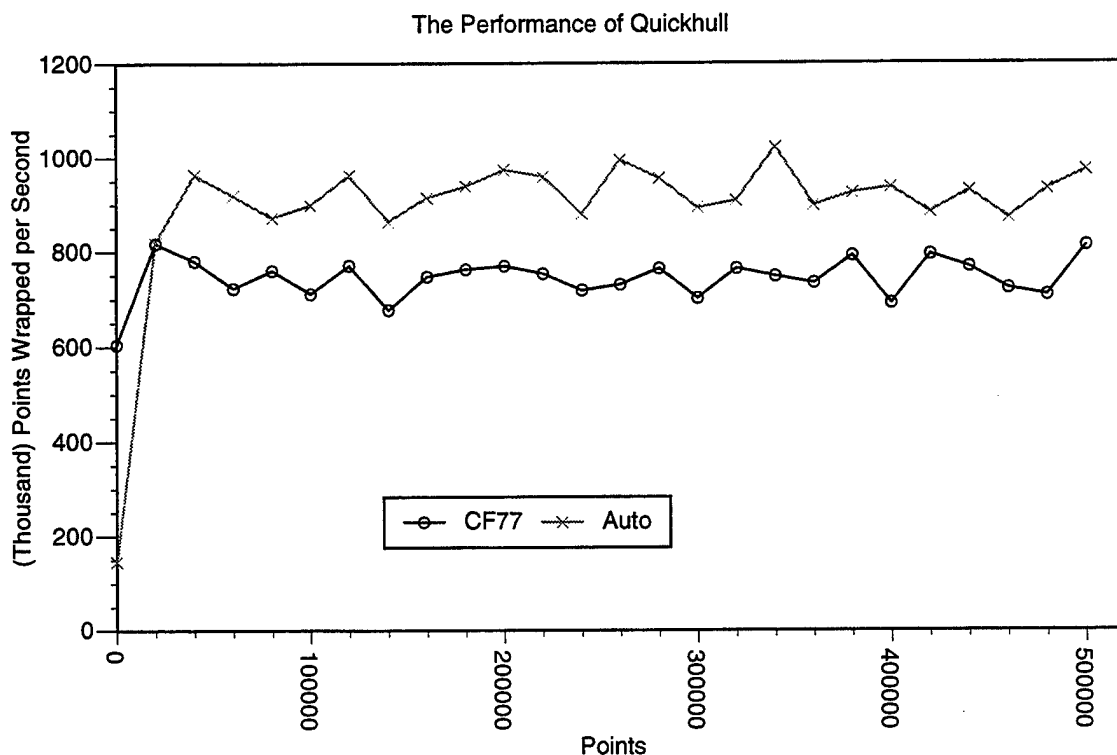
Figure 8.38 Performance of quickhull algorithm.

The last two loops in the merge phase of this algorithm are problematic, from a performance point of view. Control embedding nests the loops, which then have to be flattened. However, the loop index values will differ for each of these loops, so that reuse of loop flattening overhead is not reusable across these two loops (or the partition loop). Furthermore, these are relatively simple, non-recurrent loops. So CF77 does a reasonable job parallelizing the loop, though the parallelism decreases as the algorithm recurses deeper. As a result, the total speedup hovers around 1.25. The absolute and relative performances are plotted in figures 8.38 and 8.39.

This demonstrates a potential problem with loop flattening. Because we view loop flattening as a general control flow transformation, the overhead is more expensive in contexts like the control embedding for recursive subroutines than some alternatives. One alternative is to simply note that embedding control will segment all operations in the subroutine body. More optimized versions of segmented operations can then be employed.
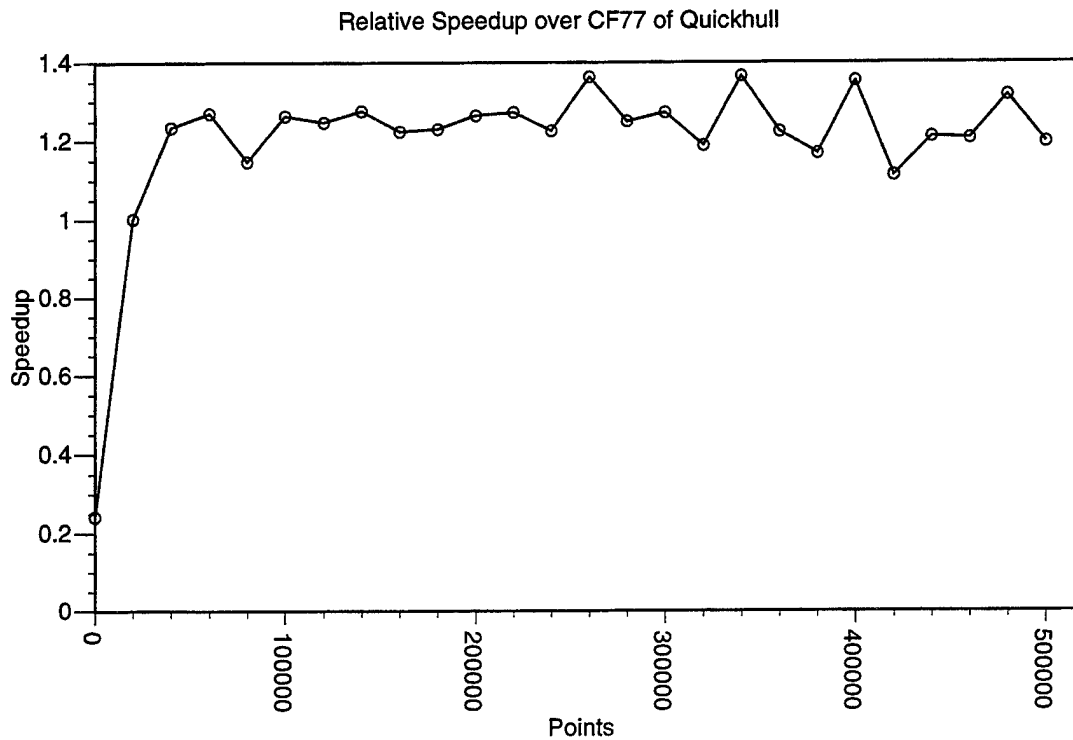
Figure 8.39 Relative speedup of quickhull algorithm.

8.5 Comparison With NESL

NESL is a high-level parallel language with support for expressing nested parallelism [18]. It allows the concise expression of many algorithms in which nested parallelism is inherent. The language is compiled to an intermediate language, whose interpreter has been ported to numerous architectures, including the Cray C90. Many of the primitives that NESL compiles to are similar or identical to those that our compiler generates, though the types of reductions and scans it can compute are limited. The code for a stable quicksort is shown below:

```
function qsort(a) =
   if (#a < 2) then a
   else
      let pivot = a[0];
         rest = a->[1:#a];
         less_greater = split(rest,{e > pivot: e in rest});
         result  = {qsort(v): v in less_greater};
      in result[0] ++ [pivot] ++ result[1];
```

We have plotted the performance of the stable quicksort as coded in both NESL and Fortran (and subsequently compiled by our compiler) on a single vector processor of the C90 in figure

Performance of Stable Quicksort for NESL and Auto Parallelization
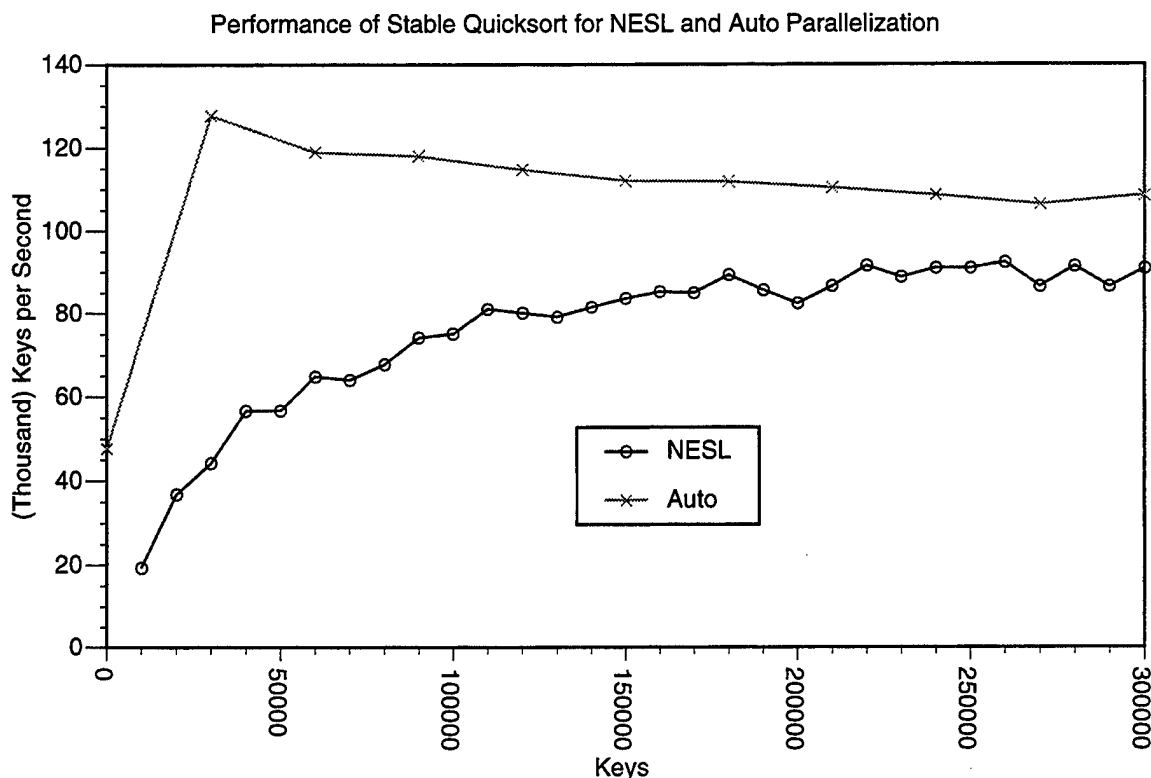
Figure 8.40 Performance of stable quicksort in code automatically parallelized and code written in NESL on a single vector processor.

dation in the NESL code. This is primarily due to the interpreter overhead of the intermediate language, VCODE [16]. The intermediate language is interpreted by a VCODE interpreter compiled for a range of architectures. The interpreter adds a constant amount of overhead to each operation executed. Furthermore, since the granularity of the intermediate code is at the level of parallel operations, there is not much optimization by loop fusion to, among other things, exploit Cray architectural features, like chaining of vector operations. However, this particular disadvantage is also true of our compiler, where we have expended little effort to optimize in this fashion across parallel operations.

8.6 Space Overhead

The compiler was not developed with memory usage optimization in mind. The schemes we use for the recurrence parallelization technique and the two control structure transformations introduced in this dissertation allocate memory fairly liberally, though in the cases of loop flattening and control embedding, the techniques of section 7.6 attempt to mitigate this. Furthermore use of more efficient representations, such as bit vectors in place of integer vectors,

might make a difference. It is worth discussing the memory usage of the recurrence parallelization technique since it introduces template variables, which may, in many cases be difficult to reuse.

For a recurrent loop of length n, the space used by the k template variables is simply kn. Compared to explicit encodings of the recurrences, the space overhead amounts to one of these template arrays. For example, if we do template variable computation "in place", this only saves space for one template variable. If the reduction or scan intrinsically requires that multiple values be computed in the combining tree, the number of template variables will only reflect this. An explicitly parallel encoding of the scan will not necessarily improve the memory requirements of the operation. For example, consider the linear recurrence of section 4.1.3.1. We derive a method which uses two template variables in computing the composition. Note, however, that a first order linear recurrence generally requires two values to be propagated in an explicit parallel prefix or reduction implementation. We can reuse the space for the value being computed, the recurrence variable, in the case of a scan operation. Thus, we save at most one memory slot over the templatized version. So the space overhead is typically n for each reduction or scan of length n generated. We may amortize this overhead between scans and reductions by reuse of template variable space.

8.7 Early Implementation Experiences

The recurrence parallelization technique has also been implemented and used to generate code for the iWarp parallel computer. The iWarp is a 8x8 grid of LIW computation cells with tightly integrated, high-speed communication paths [22]. We developed general reduction code templates for block-wise distributed arrays.

Timings from an automatically extracted addition reduction (referred to as Sum) on 64 processors are shown in figure 8.41, along with the performance of a serial implementation and the performance of the intrinsic. Despite the additional overhead of a function application step and broadcast, the slopes of the automatically extracted code and the intrinsic addition reduction operation are fairly close. The relative speedup efficiency of the automatically generated code (computed as *automatic speedup/intrinsic speedup*) for 64 processors converges to over 75%. Figure 8.42 displays speedup vs. number of processors for the sum, maximum, and
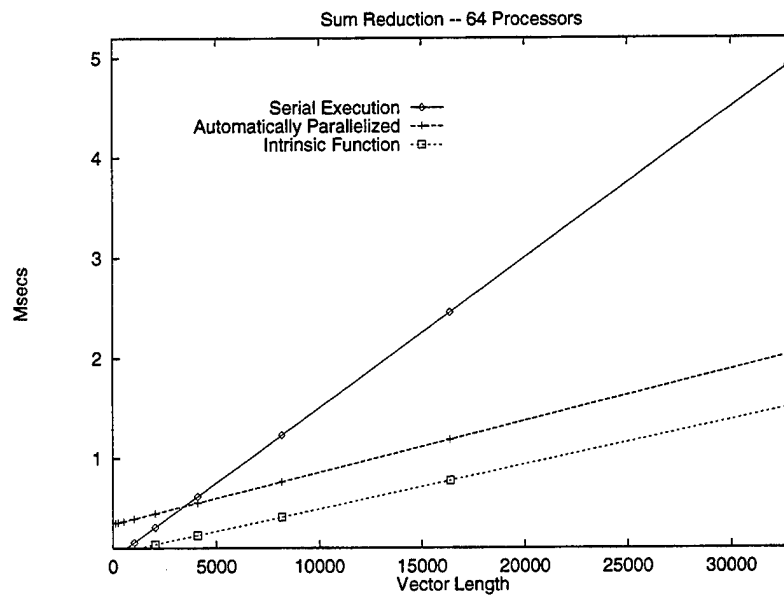
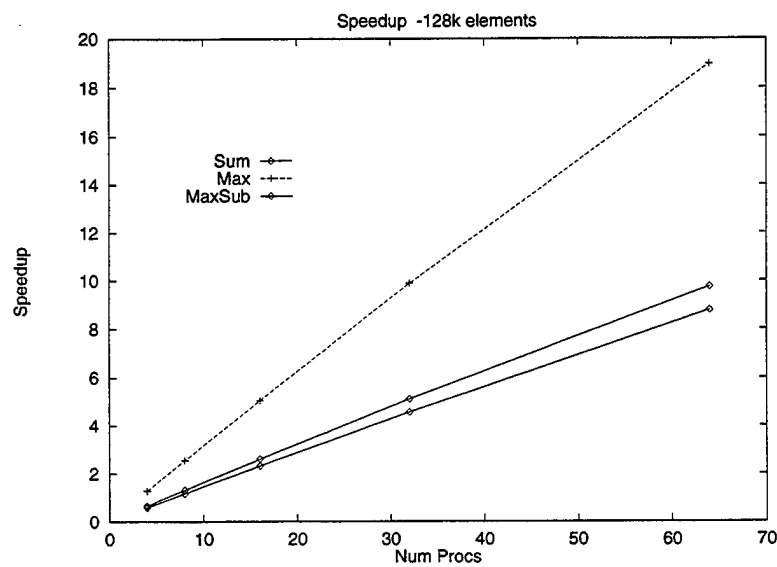*Figure 8.41 Timings of sum reduction on 64 processor iWarp array.*

Figure 8.42 Speedup of various reductions on differing iWarp array configurations.

maximum subsequence reductions. The speedup scales linearly with the number of processors.

8.8 Performance Observations

The factors affecting performance of our compiled code are numerous. Program parameters such as size, average row length for sparse matrices, and key density for combining-send are important considerations. The performance graphs we have presented here give a good indication of the type of performance one can expect from such variances in those parameter spaces. There are also less predictable factors such as data value distributions that we have attempted to mitigate by performing multiple test runs over different data sets. These factors more or less impact the performance of the particular primitives parallelized, but they do not reveal much about the compiler choices made in parallelizing the code.

Certain compiler decisions and strategies chosen here have a great impact on the overall performance. This has lead us to make several observations about the performance results here and our experiences in building this compiler:

- **Loop flattening overhead is worth optimizing.**

  The loop flattening overhead is primarily responsible for the lower relative performance improvements in the divide-and-conquer algorithms. We have optimized the overhead by parallelizing and selectively eliminating those portions of the loop index computation which are unnecessary. However, there are opportunities for reuse of this overhead that have bigger overall payoffs.

- **Reuse is important.**

  The loop flattening overhead, as well as the SPINETREE structure construction phase of the combining-send and multiprefix templates, can be reused over multiple invocations of the compiled primitive or loop under certain circumstances. In particular, if the code parallelized is embedded within another loop, and the factors affecting the loop indices (such as the sparse matrix shape) or the mappings created by the combining-send operation are invariant with respect to that loop, the code can be hoisted out. Furthermore, for multiple flattened loops or combining-send/multiprefix operations which have similar shape and/or mapping factors, the respective overheads can be recycled between them.

  The compiler performs this optimization for loop flattening overhead and combining-send and multi-prefix overhead. However, in cases where the applicability of these optimizations is limited, such as the 'quickhull' test program, the performance suffers, though it still outperforms the native compiler's performance.

- **Loop flattening manages trade-offs in granularity and load balancing.**

  Loop flattening, by effecting a flattening of nested parallelism, effectively eliminates the trade-off between granularity, load balancing, and run-time system complexity in parallelizing irregularly nested control structure. However, it is important to note that the flattening of parallelism is only conceptual from this compiler's point of view. The compiler makes note of where loop flattening is applied so that future target architectures can choose alternative implementation strategies for the particular segmented operation derived from the code. That is, flattening works on our current architecture, but the segmented operation can also be executed by parallelizing across segments or within segments depending on the target architecture (and data conditions).

- **Embedding incurs little performance cost.**

  Based on the relative speedups of the 'segmented partition' program over a range of partition counts and the divide-and-conquer style programs, one can see that loop flattening is the primary constraint on performance. In relation to loop flatteing, embedding of control is a relatively cost-free transformations from a performance point of view. This is reinforced by profiles of the parallelized code, which reveal relatively insignificant amounts of time spent in embedding overhead.

  On the other hand, the space overhead control embedding is higher. This makes sense conceptually since multiple activation records (all the activation record in a particular level of the tree) are effectively being stored in local variables expanded in rank by the embedding process. This cost is not unreasonable, since we are conceptually parallelizing across multiple function or subroutine calls, whose activation records must exist anyway. In fact, the cost might be a little lower since we do not necessarily apply expansion to all local variables in embedding.

- **Recognizing the right parallel primitive(s) is important.**

  The algorithms in this dissertation which employ nested control structure often benefit greatly from the sum total of the compiler passes in this dissertation. The abilities to generally parallelize recurrences and flatten nested parallelism in a serial language setting are each individually important. However, the two taken together allow compilers to generate what we believe to be the best algorithmic choice for parallelization of these algorithms: segmented operations. This is borne out by the positive performance results relative to the CF77 compiler, despite the relatively unoptimized quality of the primitive implementations

163

our compiler uses. In other words, the algorithmic quality of the parallelized code out-weighs the quality of the particular implementation of a primitive relative to alternative parallelization strategies (i.e. the CF77 compiler's parallelization without flattening). The algorithms automatically parallelized by our compiler scales better to multiple processor configurations than the code parallelized by the CF77 compiler, as evidenced by the CSR and CSC sparse matrix-vector multiplication examples.

8.9 Opportunities for Performance Improvement

A number of opportunities exist for improving the performance of code generated by our compiler. These are primarily post-parallelization optimizations. That is, the code is still parallelized using the techniques in this dissertation, but is further optimized afterwards.

- **Interprocedural reuse of loop flattening overhead**

  As mentioned in the previous section. The goal here would be to reuse the loop index computation portion of the loop flattening overhead incurred during divide-and-conquer algorithms. Recomputing the entire loop index set is redundant, since less expensive operations can be used to recycle that computation.

- **Reuse of combining-send and multiprefix structure**

  This entails the reuse of the SPINETREE structure that is built as part of our combining-send and multiprefix operations. This may also involve reusing sorted sequences in alternative combining-send schemes. One scheme for executing a combining send is to perform a stable sort of the source array using the indices as keys. Then a simple segmented reduction or scan can be performed to compute the combining-send and multiprefix operations, respectively. A scatter or permute of the results might be necessary after this step. The advantage of this approach is in contexts where reuse is likely to be high. Though the sorting step is expensive, if we can reuse it often enough the potential advantage of approaching reduction and scan performance over using the SPINETREE structure may be worthwhile.

- **Scaling to multiple processors**

  While we have written multiple processor versions of our basic reduction and scan templates and we can generate those, the problem with this is our back-end, the Cray CF77 compiler. The CF77 compiler does **very well** at vectorization, which is the primary reason

we chose to compare our compiler against it in the single vector processor case. However, unless loop nests are present, the CF77 compiler does not simultaneously vectorize and parallelize well (the sparse matrix-vector multiplication kernels were exceptions since there was a loop nest). So, to exploit parallelism in those parts of the code that our front-end was not directly responsible for parallelizing, we would have to build a better mechanism for parallelizing and tasking those pieces of code that are neither recurrent or irregular. Unfortunately, this is outside the scope of this thesis.

It is important to note that the operations that we parallelize, such as reductions, scans, combining-sends, and multiprefix along with their segmented variations, do scale to multiple processors very well. The CSR example, in which the multiprocessed template conveniently computed the entire loop nest, illustrates the performance of a segmented reduction on multiple vector processors. It is also important to note that the speedup scaling is true on a wide range of parallel architectures.

- **Efficient representation and computation of segments**

The representation of segmentation in the recurrences we parallelize is implicit in the operators we derive for use by parallel reductions and scans. The computation involved in computing with segmentation typically includes two multiplications and an addition for each element, in addition to the operator in the reduction or scan. This an artifact of the way the underlying analysis treats loop invariant conditions. There are more efficient ways of computing where segments begin and end that involve conditionals. A conditional expression in the combining operator used to select where values should be inserted in a segmented operation can be generated with a slight modification to the underlying analysis.

The compiler has opportunities to reuse segment-related computation across multiple segmented operations derived automatically from loops with identical loop bounds. One way in which this might be effected is to insert explicit representations of the parallelized code, rather than subroutine calls to specialized templates. That way, loops across several reductions or scan can be fused, creating opportunities to eliminate redundancy.

Space efficient representation schemes for segmented operation are also possible. Bit vectors and segment-length vectors are two such candidates. It is possible that the analysis could be extended to manage such structures.

8.10 Review

This chapter presented the results of compiling a range of loops and algorithms using the techniques presented in this dissertation. For simple recurrences, the performance of our general code templates are shown to be competitive with the optimized versions used by CF77. For more complex recurrences, the code our compiler generates performs significantly better since CF77 has limited ability to parallelize such recurrences. The relative performance improvements gradually decrease with the application of loop flattening and control embedding, especially in cases where the loop flattening overhead cannot be eliminated or amortized over multiple instances of the loop.

Chapter 9

Related Work

Some components of this thesis have precedent, though the particular combination of transformations and the power they display in a compilation system is unprecedented. In this chapter, we discuss work related to the individual components of the compiler. We primarily focus on compiler transformations and analysis. Excellent related work references on algorithms and parallel primitives may be found in the relevant source literature [15][18][32][49][62][71][76][77][96].

9.1 Automatically Parallelizing Recurrences

Automatic recognition and efficient solution generation for recurrences from serial code has mostly been limited to finding a pattern that matches a known recurrences and then using one of a library of fast solutions to solve it. These methods are limited by several factors:

- They are limited by the set of recurrences programmed into the compiler.

- They have limited ability to solve recurrences involving arbitrarily nested conditional operators.

- They are dependent on the syntactic quality of the source code.

Pinter and Pinter's algorithm [66] does well at parallelizing recurrences with simple filtering (non-dependent) conditionals. However, they depend on the syntactic quality of the source code rather than the semantic content. They pattern match on the dependence graph to find the recurrence operator. Sensitivity to source code forms is generally a problem for the pattern matching methods employed in many commercial compilers.

Several semantic techniques [6][50][72] have been proposed as a step toward using algebraic properties to simplify complex loop body structures, thus addressing the problem of source code quality. However, they are also limited by pattern matching for fixed set of recurrence operators. This, in turn, limits the applicability of some of the powerful control structure transformations discussed in this thesis.

Callahan [24] and Chen and Hou [27] both proposed a similar model to ours for recurrent loop execution. Callahan does not pursue aggressive symbolic analysis to automatically derive solutions, instead suggesting a pattern match against a set of core recurrences. He does suggest useful ways of combining recurrences to amortize the overhead of computing a recurrent primitive and to increase the effective granularity of operations. The mechanisms are useful to us because the particular components manipulated are essentially identical to those we automatically derive.

Chen and Hou make the observation that the associativity of function composition can be exploited on functions on finite sets. The basic idea is to compute all possible composite functions, which is tractable since the domains and ranges they consider are small and finite. Though interesting, this is not particularly useful for more general computation on integers and real numbers. They have not, to our knowledge, used this observation in a compiler.

Mou et al. [63] characterize whether a recurrence is parallelizable based on topological properties of the recurrence. Those topological properties include whether an unfolded (similar to unrolling a loop once) recurrence is conservative in the sense of preserving its topology. This is analogous to our scheme for composing and testing the result for inclusion in the original functions class. However, their work is essentially limited to those recurrences computable by reductions and scans, and whose forms are essentially linear. The reason is that they had not developed a framework for generally building the necessary associative operators.

More recently, Rinard and Diniz [73] have developed a mechanism to use *commutativity analysis* to parallelize programs. The idea is that if a collection of operations commute, then they can be executed in any order, and thus parallelize. The kind of analysis they employ to uncover commuting operations is similar to ours, however the range of operations which commute is considerably smaller. They exploit the encapsulation properties of C++-based programs to parallelize such graph-based algorithms as Barnes-Hut [12].

168

Induction variables comprise a subset of the type of recurrent computations discussed here. Induction variable computations typically have a closed form over the loops index variables (though monotonic induction variables do not). Since the compiler can typically eliminate the variable by substituting the closed form expressions, using reductions or scans to compute induction variables is usually contraindicated because of the higher overhead. The problem of identifying various induction variables has been explored in some depth, though typically pattern matching on various program representations is used. The most recent work relies on the Single Static Assignment (SSA) graph [94], or its variants [87].

The nature of the analysis is similar to abstract interpretation [3][35]. This is an analysis technique whereby the program is executed in an abstract domain much simpler than the usual semantic domain programs run in. The idea is to capture some property of the program based on the meaning of language features in that abstract domain. For example, a traditional data-flow analysis framework supports particular abstract interpretations of programs for various types of data-flow problems. Though very different, the modeling functions in our analysis comprise an abstract domain in which we execute a limited type of program and language feature: non-nested recurrent loops.

9.2 Flattening Loop Nests

Loop flattening is generally not a new concept. Loop coalescing [67] and loop collapsing [52] are both similar transformations for regular loop nests. Loop flattening provides essentially the same benefits in terms of load balancing and availability of parallelism, but differs significantly in key areas. Subscript simplification in the regular loop transformations may only involves simplifying algebraic expressions, while indirection may defeat this approach for irregular loop nests.

Hanxleden and Kennedy [41] proposed a general notion of loop flattening to facilitate parallelization of nested irregular loops. We benefit from the presence of the advanced recurrence transformations in our compiler, which, in concert with loop flattening, allow the recognition of segmented reductions, segmented scans, and combining-sends. Also, we attempt to optimize the potentially costly overhead of computing original loop indices.

However, these works, with the exception of the loop coalescing work, were all preceded and subsumed by the parallelism flattening work of Blelloch [14]. Our work translates this work

into an automatically parallelizing compiler context, with the necessary analysis and source code transformation to support such a transformation. One primary difference is that our simple transformation does not have the breadth of general flattening of parallelism. Another is that we choose an *a priori* flattening scheme rather than a flattening of composed, parallelized operations. The primary reason for flattening control structure rather than parallel primitives is the difficulty for the compiler in parallelizing complex codes that include mixes of recurrent and non-recurrent codes in multiply nested loops. Contrasted with the simpler, yet still powerful, loop flattening approach, the relative cost of engineering the compiler in such a way to support the more general approach seemed high. We pay a price of having potentially more expensive overhead, as well as more complex source code presented to the parallelizer. However, this last issue works to our advantage because of the nature of our recurrence parallelization technique.

In flattening arbitrarily nested parallelism, we are performing a function similar to one performed by compilers for some high level parallel languages. One such language is Nesl [18], a portable, nested data-parallel language. A phase of the Nesl compiler flattens explicitly specified expressions of nested parallelism into non-nested parallelism.

A good deal of work has been done in parallelizing irregular array access patterns in the form of array accesses with multiple levels of indirection [31][42]. However, this work is somewhat orthogonal to the issues discussed in this paper, though some of the data-flow analysis problems can be applied here as well. Lucco [56] presents dynamic scheduling algorithms for wider range of irregular code. Such scheduling methods might be effective mechanisms for parallelizing the primitives derived by the compiler analyses presented in this dissertation.

Blelloch [15] demonstrated that segmented reductions and scans were useful and efficient parallel primitives for a broad range of algorithms. Blelloch et al. [19] and Sheffler [76] found applications of these segmented parallel primitives, merges, and combining-sends to sparse matrix algorithms. Other parallelized sparse matrix algorithms are discussed in [19][51].

170

9.3 Control Embedding

To our knowledge, there has been no other work in automatically parallelizing divide-and-conquer algorithms in serial, imperative languages. However, the transformation does have roots in a variety of compiler transformations for a variety of language and programming paradigms.

The idea of embedding control structure in function calls has precedence in work on loop embedding [39]. Loop embedding exposes code to parallelization by embedding surrounding iterative control structure (loops) in subroutine calls. What our embedding transformation achieves is more general in the sense that we also embed surrounding conditional structure and handle recursions (i.e. we compute, in code, the fix-point of the transformation when embedding in a recursive subroutine). However, we apply the transformation more narrowly to recursive subroutines.

The closest transformation to control embedding for recursion is not found in the automatic parallelization world. The Nesl compiler [18] folds expressions of nested parallelism into embedded function calls, though its task is made considerably easier by the source language. The compiler can then apply the general parallelism flattening scheme.

9.4 Other Parallelization Techniques

There has been a considerable effort in transforming regular loop nests to expose loops without loop-carried dependences to parallelization [91][93]. These are effective at transforming superficially recurrent loops within loop nests. The more general loop transformation frameworks are also quite effective at being adapted to different parallelization styles, as well as being extended to solving other important problems in compiling for high performance, such as optimizing data layout [7] and improving data locality [92]. However, none of the programs or loops considered in this dissertation can be effectively parallelized by such techniques. The presence of irregularity or lack of nesting of non-recurrent loop is the stumbling block.

Chapter 10

Conclusion

10.1 Summary

The primary goal of this work is to move beyond dependence analysis and pattern matching based approaches to parallelization, specifically as they relate to parallelizing recurrent and irregular computation. The main claim of this dissertation is that recurrent and irregular code can be automatically mapped into useful higher level parallel primitives in a general and reliable manner.

The thesis of this dissertation was proven through the development and testing of several new analyses and transformations. Rather than just comprising a collection of unrelated compiler transformations, the design and selection of the new compiler passes complement each other well. Each individual transformation is useful, but the combination of them has a significantly larger impact. The primary reason is that our recurrence parallelization technique is robust and flexible.

We have designed and implemented a technique which uses the following concepts and methods to automatically extract parallelized code for recurrences:

- A model for recurrent loop bodies which is always associative.

- Symbolic substitution of expressions.

- Linear relation feasibility testing to simplify complex conditional structures.

- Logic minimization techniques to reduce conditional nesting structure.

- A specialized unification algorithm which abstracts out subexpressions of loop invariant values.

We distinguish our technique from existing techniques with the following:

- We are able to find solutions for a broad class of recurrences by actually extracting an efficient associative operator from the source code. We rely on the analytical abilities of the compiler rather than a fixed collection of recurrences.

- We handle conditional operators embedded within recurrences in a general way. The ability to deal with conditionals in a robust manner is critical for handling more complex control structure contexts.

- Our model of recurrences is more general than prior approaches. The analysis here can be extended to other forms of recurrence, such as combining-sends and list-prefix operations.

We use a basic loop flattening transformation that facilitates the parallelization of irregular loop nests and provides a basis for recognizing more sophisticated parallel primitives. The basic idea is to create a single, non-nested loop that emulates the execution of the original loop nest. This is achieved by first computing the original loop nest's index sets. Then, by creating a non-nested loop with a trip count equal to the total sum of inner loop trip counts, the precomputed index sets are used to decide which point in the original loop the flattened loop should execute. The transformation has the following benefits:

- The application of most existing parallelization transformations, as well as our recurrence parallelizing technique, to the loop nest *in toto* by applying the transformations to the *flattened* loop. Artifacts of the loop flattening transformation include complicated conditional structures, which the recurrence parallelization technique must be able to deal with.

- The parallelization of the index set computation. This allows the practical use of loop flattening in a compiler by attacking the remaining artifact of the transformation: the index set computation. Also, this leads to some intriguing future possibilities for extracting other sophisticated algorithmic idioms from irregular loops.

- The amortization of the index set computation over repeated executions of the loop nest. For sparse matrix-vector multiplication, this is analogous to the preprocessing steps of many existing parallel libraries, most of which are applied once for repeat multiplications of the matrix.

Parallelizing irregular loops *in toto* is important for the following reasons:

174

- Load balancing the assignment of outer loop iterations is complicated by unpredictable inner loop trip counts.

- Inner loop trip counts may not be sufficiently large to make parallelizing the inner loop body worthwhile.

Embedding control for divide-and-conquer style recursion allows the compiler to exploit both intra- and inter-procedural parallelism. This parallelism results from using different procedural contexts to work on independent subproblems of original problem. The problems in divide-and-conquer algorithms closely mirror those of irregular loop nests (because they both are expressions of arbitrarily nested parallelism [14]).

- Ever decreasing amounts of parallelism are available in partitions as they are repeatedly subdivided. However, the amount of parallelism across all sibling partitions remains relatively constant with reasonable partitioning functions. Thus, parallelizing across all partitions simultaneously inures the parallelized code to smaller partition sizes.

- Partition sizes may vary depending on the strategy. This may result in unpredictable and unbalanced load situations. Parallelizing across all partitions simultaneously creates the greatest opportunity for avoiding this problem. Furthermore, the number of partitions at early stages is relatively small (the reverse of the situation with partition sizes). Parallelizing between partitions only would result in a lack of parallelism at early stages of execution. Parallelizing across all partitions eliminates the need to consider the relative amounts of parallelism in the partitions and across the partitions at various stages of computation.

Using these three compiler techniques make algorithmic sense, but they also interact well in an operational sense. The artifacts created by loop flattening are well-suited for the associativity analysis of the recurrence parallelization technique, since loop flattening produces nested conditional structure and indirect accesses which look like recurrences, but eliminates nested iterative structure. Both of which are, in turn, well-suited to follow control embedding, since its control embedding yields nested and, typically, irregular loops. Each compiler pass in turn generates exposes more code to parallelization for the next pass of the compiler. The net effect is that we have a system which enables the parallelization of many more useful algorithms than the sum of each transformation's individual impact. The performance improvements relative to the Cray C90's native CF77 compiler were good across a range of programs.

Among the lessons learned from building this compiler and analyzing the code generated was that details matter. The new transformations, in choosing superior parallel algorithms, account for significant performance improvements, but how they were engineered had a large impact upon the performance. Careful selection of code templates for recurrences and mechanisms for tracking code transformation through the phases of the compiler were very important to both the performance of the compiler and the code it generated. Because some of the higher-level primitives incurred relatively high computational and memory overhead, reuse was also an important factor in the performance of many of the applications parallelized here. In section 8.9 and future work section (10.2) of this chapter, we discuss further extensions to the compiler to improve performance.

10.2 Future Work

Based on our experiences, we believe that we have reached a point of diminishing returns in parallelizing further recurrent primitives in Fortran. However, we believe that there are significant opportunities to apply this work and extensions thereof to parallelizing C and C++. The primary opportunity here is the use of pointer-based, recursively defined data structures. Recurrent operations on linked lists, graphs, and trees can often be parallelized using a parallel list-prefix algorithm [96] or parallel tree contraction [62]. An important step toward to achieving this is the evolution of dependence analysis for pointer-based structures [48][90].

The compiler currently uses fixed code templates for computing the various recurrent primitives. However, there are opportunities to exploit both similarities and the structure of those templates. First, the compiler uses generally using a pattern matching scheme on the dependence graph and index expressions to determine the type of parallel primitive to use. It may be possible just to have a higher level prototype of a reassociating, recurrent primitives which the compiler can specialize to an appropriate primitive based on the target variable and its indexing scheme (if any). Furthermore, rather than generate specialized library calls for each parallelized loop, as we are essentially doing, there are opportunities for optimizing the structures of the recurrent primitives in context. For example, under certain circumstances there are reusable portions of a multiprefix operation (the SPINETREE structure or sort, depending on the implementation we use). These pieces of code can be moved outside of surrounding loops to eliminate redundancy.

176

A fundamental problem in compiling irregular loop nests is finding a way to handle the index set computation. A fixed, parallelized preamble for computing index sets for segmented loop nests was introduced in this paper. It is possible to optimized this preamble on a case-by-case basis, to embed it in other parallelizable loops, and so forth. However, the more general parallelism flattening scheme might afford the compiler better opportunities to optimize in this manner.

Irregular loop nests come in many flavors. We have briefly studied strategies for compiling irregular loop nests that include while loop and conditioned loop exits. Many of these loops should be parallelized using parallel merging and pattern matching primitives.

The overhead of computing index sets for loop flattening is a source of concern. A more general interprocedural partial redundancy elimination scheme would be useful for hoisting loop flattening and combing-send overhead (as mentioned in section 10.2) out of subroutines. There also may be more efficient mechanism of representing and partially recycling index set computations, especially for loops creating by control embedding of a divide-and-conquer algorithm.

As the state of the art in dependence analysis for pointer-based structures and interprocedural analysis advances, we expect to find wider applicability of these techniques to languages like C and C++. We think this would be useful for many tree- and graph-structure traversal algorithms, as well as the divide-and-conquer algorithms presented here. The control structure transformation presented here can straightforwardly be applied to such codes. The difficulty in adapting this work to such domains is in managing such complex data-structures for parallel execution contexts. However, we suspect that lessons learned from other projects involved in parallelizing C and C++ can be applied here [9][57].

The application of alternative models and more sophisticated reasoning mechanism in a compiler leads to many intriguing possibilities for the future of compiler design. Parallelizing compilers now routinely include mechanisms for efficiently deciding linear inequalities, which opens the door for sophisticated program analysis. For example, rather than being largely restricted to less accurate heuristic dependence tests, more exact (and ever faster) analyses have been developed [59][69]. An open question is whether general analysis frameworks can be built to support the representation and transformation of new program models. A crucial issue to the relevance of this question is the relative expense of such a framework and

analysis. Both the potential performance payoffs and the applicability of such a framework to a wide range of analyses should be considered. In light of this, we hope the analysis presented here, as well as elsewhere [73], to be promising first steps toward justifying the expense.

10.3 Concluding Remarks

This dissertation bridges the gap between the work of the parallel language and model community and the parallelizing compiler community. Powerful primitives, such as reduction, scan, combining-send, and multiprefix operations, and support for nested parallelism have been extremely effective at providing a concise language in which to expression relatively complex algorithms. Unfortunately, the automatic parallelizing compiler community had not adequately addressed such expression in serial code, assuming them to be either intractable or that stop-gap measures were sufficient.

The philosophy of our approach is that recurrence parallelization must be extended beyond what had been done prior this work, but in a well-founded manner. We have developed general and robust compilation techniques for automatically parallelizing recurrent loops into reduction, scan, combining-send, and multiprefix operations. These techniques are flexible enough to deal with other important components of the compilation process.

The ability of the recurrence analysis to handle complex conditionals has enabled us to develop effective compilation techniques for parallelizing irregular loop nests. These techniques are fully compatible with both the underlying recurrence parallelization technique and traditional dependence based approaches. Furthermore, we have also developed techniques and analysis to discover expressions of nested parallelism in serial encoding of divide-and-conquer algorithms and to transform them for both parallelization and flattening. Both of these control structure transformations are fully compatible with underlying techniques for parallelizing recurrent loops and non-recurrent loops.

The net effect of these transformations is a radical improvement in the range of programs that parallelizing compilers can parallelize effectively. This compiler technology brings useful parallel primitives and building blocks to users of serial languages. We have proven this by presenting performance results of compiling programs and loop kernels from a range of

benchmark suites and application domains. Our compiler beats a good native vectorizing compiler for the C90 in nearly all of these cases. More importantly, the presentation of such automatic parallelization results for serial, imperative languages is unprecedented.

Chapter 11

Bibliography

[1] Scientific Libraries Reference Manual. Cray Research, Inc. Publication SR-2081.

[2] UNICOS Performance Utilities Reference Manual. Cray Research, Inc. Publication SR-2040.

[3] S. Abramsky and C.L. Hankin. *Abstract interpretation of declarative languages*. Ellis Horwood, 1987.

[4] A. Aho, J. Hopcroft, and J. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, 1974.

[5] A. Aho, R. Sethi, and J. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, 1988.

[6] Z. Ammarguellat and W.L. Harrison III. Automatic recognition of induction variables and recurrence relations by abstract interpretation. In *Proceedings of Sigplan 1990*, Yorktown Heights, NY, 1990.

[7] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 166-178, Santa Barbara, CA, July, 1995.

[8] Applied Parallel Research, Inc. *Forge Magic/DM: User's Guide*. Version 1.0, 1993.

[9] J. Arabe, A. Beguelin, B. Lowekamp, and E. Seligman. Dome: parallel programming in a distributed computing environment. In *Proceedings of International Conference on*

Parallel Processing, Honolulu, HI, April 1996.

[10] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. In *ACM Computing Surveys*, Vol. 23, No. 3, pp. 345--405, September 1991.

[11] R. Ballance, A. Maccabe, and K. Ottenstein. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pp. 257-271, June 1990.

[12] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm.*Nature* 6096(324):446-449, December 1986.

[13] J. Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[14] G. Blelloch. *Scan primitives and parallel vector models*. Ph.D. Dissertation, Massachusetts Institute of Technology Laboratory for Computer Science, 1989.

[15] G. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, C-38(11):1526-1538, November 1989.

[16] G. Blelloch and S. Chatterjee. VCODE: A data-parallel intermediate language. In Proceedings Frontiers of Massively Parallel Computation, October 1990.

[17] G. Blelloch, S. Chatterjee, J. Hardwick, M. Reid-Miller, J. Sipelstein, and Marco Zagha. CVL: A C vector library. School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-93-114, February 1993.

[18] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. San Diego, CA, May 1993.

[19] G. Blelloch, S. Chatterjee, and M. Zagha. Solving linear recurrences with loop raking. In *Proceedings Sixth International Parallel Processing Symposium*, March 1992.

[20] G. Blelloch, M. Heroux, and M. Zagha. Segmented operations for sparse matrix

computation on vector multiprocessors. Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-93-173, August 1993.

[21] G. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing,* 8(2):119-134, February 1990.

[22] S. Borkar, R. Cohn, G. Cox, S. Gleason, T. Gross, H.T. Kung, M. Lam, B. Moore, C. Peterson, J. Pieper, L. Rankin, P.S. Tseng, J. Sutton, J. Urbanski, and J. Webb. iWarp: An integrated solution to high-speed parallel computing. In *Supercomputing '88,* pp. 330-339, November 1988.

[23] Robert K. Brayton. *Logic minimization algorithms for VLSI synthesis.* Kluwer Academic Publishers, Boston, 1984.

[24] David Callahan. Recognizing and parallelizing bounded recurrences. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Processing,* Santa Clara, CA 1992.

[25] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. ACM Computing Surveys, 21(3):323-357, September 1989.

[26] K.M. Chandy and C. Kesselman. Compositional C++: compositional parallel programming. In *Languages and Compilers for Parallel Computing, 5th International Workshop Proceedings,* New Haven, CT, August 1992.

[27] R.J. Chen and Y.S. Hou. Non-associative parallel prefix computation. In *Information Processing Letters,* 44:91-94, 1992.

[28] Shyh-Ching Chen and David J. Kuck. Time and parallel processor bounds for linear recurrence systems. *IEEE Transactions on Computers,* C-24(7), July 1975.

[29] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proceedings Supercomputing '93,* pp. 262-273, Portland, OR, November 1993.

[30] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems.* 13(4):451-490, October 1991.

[31] R. Das, J. Saltz, and R. von Hanxleden. Slicing analysis and indirect accesses to distributed arrays. Technical Report CS-TR-3076, University of Maryland, May 1993.

[32] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1986.

[33] R. J. Duffin. On Fourier's analysis of linear inequality systems. *Mathematical Programming Study no. 1,* North-Holland, 1974.

[34] John T. Feo. An analysis of the computational and parallel complexity of the livermore loops. *Parallel Computing,* 7:163-185, 1988.

[35] A. Field and P. Harrison. *Functional Programming.* Addison Wesley, 1988.

[36] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation,* pp. 135-146, Orlando, FL, June 1994.

[37] I. Foster, C. Kesselman, and S. Tuecke. The Nexus task-parallel runtime system. In *Proceedings of First International Workshop on Parallel Processing,* pp. 26-31, December 1994.

[38] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3.0 User's Guide and Reference Manual.* February, 1993.

[39] M. Hall, K. Kennedy, and K. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings Supercomputing '91,* Albuquerque, NM, November 1991.

[40] R. Halstead. Multilisp: A language for concurrent symbolic computation. In ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, pp. 501-538, October 1985.

[41] R. von Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation,* pp. 188-199, San Francisco, CA, June 1992.

[42] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler Analysis

for Irregular Problems in Fortran D. In *Languages and Compilers for Parallel Computing, 5th International Workshop Proceedings,* pp. 97-111, New Haven, CT, August 1992.

[43] J. Hardwick. Porting a vector library: A comparison of MPI, Paris, CMMD, and PVM. In *Proceedings of the 1994 Scalable Parallel Libraries Conference,* pp. 68-77, October, 1994.

[44] J. Hardwick. An efficient implementation of nested data parallelism for irregular divide-and-conquer algorithms. In *Proceedings First International Workshop on High-Level Programming Models and Supportive Environments,* Honolulu, HI, April 1996.

[45] P. Hatcher, A. Lapadula, R. Jones, M. Quinn, and R. Anderson. A production-quality C* compiler for hypercube multicomputers. In *Proceedings of the Third ACM SIG-PLAN Symposium on Principles & Practice of Parallel Programming,* Williamsburg, VA, April 1991.

[46] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.0. May 1993.

[47] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *Communications of the ACM,* 29(12), December 1986.

[48] J. Hummel, L. Hendren and A. Nicolau. A general dependence test for dynamic, pointer-based data structures. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation,* Orlando, FL, June 1994.

[49] Joseph Jaja. *An Introduction to Parallel Algorithms.* Addison Wesley, 1992.

[50] Pierre Jouvelot and Babak Dehbonei. A Unified Semantic Approach for the Vectorization and Parallelization of Generalized Reductions. In *ACM SIGARCH International Conference on Supercomputing,* Crete, 1989.

[51] D. R. Kincaid and T. C. Oppe. Recent vectorization and parallelization of ITPACKV. Technical report, Center for Numerical Analysis, The University of Texas at Austin, November 1984.

[52] C. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vec-

torizer for pipelined processors. In the proceedings of The 4th International Computer Software and Applications Conference (COMPSAC 80), October 1980.

[53] Peter M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transactions on Computers*, C-22(8):786-793, August 1973.

[54] C. Lasser. *The Essential *Lisp Manual*. Thinking Machines Corporation, Cambridge, MA, July 1986.

[55] David Levine, David Callahan, and Jack Dongarra. A Comparative Study of Automatic Vectorizing Compilers. Mathematics and Computer Science Division, Argonne National Laboratory Technical Report MCS-P218-0391.

[56] S. Lucco. A dynamic scheduling method for irregular parallel programs. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 200-211, San Francisco, CA, June 1992.

[57] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, and F. Bodin. Performance analysis of pC++: a portable data-parallel programming system for scalable parallel computers. In *Proceedings of 8th International Parallel Processing Symposium*, Cancun, Mexico, April 1994.

[58] S. Marlow and P. Wadler. Deforestation for higher-order functions. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pp. 154-165, Glasgow, UK, July 1992.

[59] D. Maydan, J. Hennessy, and M. Lam.Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[60] Message Passing Interface Forum. Draft Document for a Standard Message-Passing Interface. University of Tennessee Technical Report CS-93-214, November, 1993.

[61] Ronald E. Mickens. *Difference Equations*. Von Nostrand Reinhold Company, New York, NY, 1987.

[62] G. Miller and J. Reif. Parallel tree contraction. *Advances in Computing Research*, Vol. 5, pp. 47-42, 1989, JAI Press Inc.

186

[63] Z.G. Mou, A.J. Huang, and T. Hickey. Parallel Recurrece Transformation. In *Proceedings of the 1992 DAGS/PC Symposium*, June 1992.

[64] W. Oed. Cray Y-MP C90: system features and early benchmark results. In *Parallel Computing*, Vol. 18, No. 8, pp. 947-954, August 1992.

[65] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184-1201, December 1986.

[66] Shlomit S. Pinter and Ron Y. Pinter. Program optimization and parallelization using idioms. In *Conference Record of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 79-92, Orlando, FL, January 1991.

[67] C. Polychronopoulos. Loop coalescing: a compiler transformation for parallel machines. In *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, IL, August 1987.

[68] F. Preparata and M. I. Shamos. Computational Geometry. Springer-Verlag, New York, NY, 1985.

[69] W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. University of Maryland at College Park, Computer Science Technical Report CS-TR-2648, 1991.

[70] W. Pugh. Counting solutions to Presburger formulas: how and why. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation* , pp. 121-34, Orlando, FL, June 1994.

[71] M. Reid-Miller and G. E. Blelloch. List ranking and list scan on the Cray C-90. Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-94-101.

[72] X. Redon and P. Feautrier. Detection of recurrences in sequential programs with loops. In *PARLE '93*, Munich, Germany, pp. 132- 145, June 1993.

[73] M. Rinard and P. Diniz. Commutativity Analysis: A New Framework for Parallelization. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[74] J. Rose and G. Steele Jr. C*: An extended C Language for Data Parallel Programming. Thinking Machines Corporation Report PL87-5, April 1987.

[75] S. Saini and D. Bailey. NAS Parallel Benchmarks Results 10-95. NASA Ames Research Center Technical Report NAS-95-019, October 1995.

[76] T. J. Sheffler. Implementing the multiprefix operation on parallel and vector computers. Carnegie Mellon University, School of Computer Science Technical Report CMU-CS-92-173, August 1992.

[77] T. J. Sheffler. Match and move, an approach to data parallel computing. Ph.D. Dissertation. Carnegie Mellon University, School of Computer Science 1992.

[78] Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the Association for Computing Machinery*, Vol. 28, No. 4, pp.769-779, October 1981.

[79] H. Simon and E. Strohmaier. Amdahl's law and the statistical content of the NAS parallel benchmarks. In *Supercomputer*, Vol. 11, No. 4, September 1995.

[80] H. Stone. Parallel Tridiagonal Equation Solvers. In ACM *Transactions on Mathematical Software*, Vol. 1, No. 4, pp. 289-307, Deceber 1975.

[81] Jaspal Subhlok, James M. Stichnoth, David R. O'Hallaron, and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Proceedings of The Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, May 1993, San Diego, CA.

[82] J. Subhlok, D. R. O'Hallaron, T. Gross, P. A. Dinda, and J. Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proceedings Supercomputing '94*, Washington, D.C., November 1994.

[83] Thinking Machines Corporation. Connection Machine Parallel Instruction Set (PARIS). July, 1986.

[84] Thnking Machines Corporation. *The Connection Machine CM-5 Technical Summary*. October 1991.

[85] Thinking Machines Corporation. *CMMD Reference Manual. Version 3.0*. May 1993.

188

[86] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pp. 176-185, July 1986, Palo Alto, CA.

[87] Peng Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. Ph. D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[88] S. Vajapeyam, G.S. Sohi, and W.-C. Hsu. An empirical study of the Cray Y-MP processor using the PERFECT Club benchmarks. In *Proceedings of the 18th International Symposium on Computer Architecture*, Toronto, Canada, May 1991.

[89] D. Wall. Predicting program behavior from real or estimated profiles. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, Palo Alto, CA, June 1991.

[90] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[91] M. Wolf and M. Lam. A loop tranformation theory and an algorithm to maximize parallelism. *Transactions on Parallel Distributed Systems*, 2(4):452-470, October 1991.

[92] M. Wolf and M. Lam. A datalocality optimizing algorithm. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pp. 30-44, June 1991.

[93] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, Cambridge, MA, 1989.

[94] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[95] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *International Journal of Parallel Programming*, Vol. 28, No. 2, April 1987, pp. 137-178.

[96] J. C. Wyllie. *The Complexity of Parallel Computations*. Ph.D. Dissertation, Computer Science Department, Cornell University, Ithaca, NY, 1979.

[97] P. Yang, J. Webb, J. Stichnoth, D. O'Hallaron, and T. Gross. Do & Merge: Integrating parallel loops and reductions. In *The Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, Oregon, August 1993.